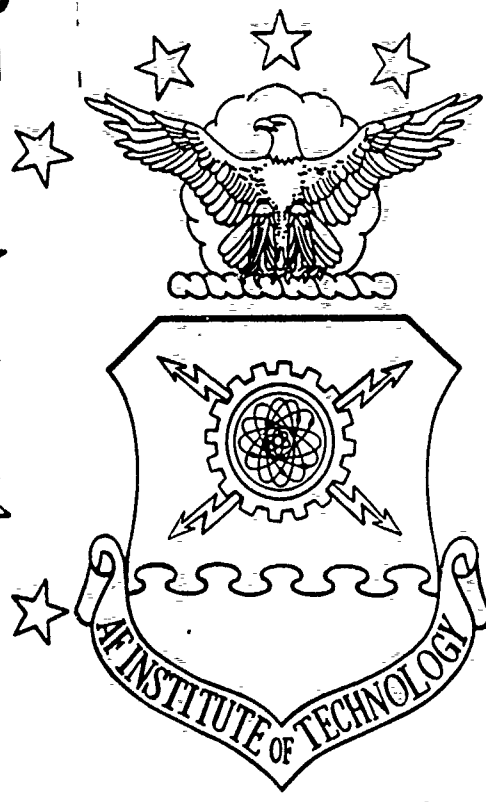
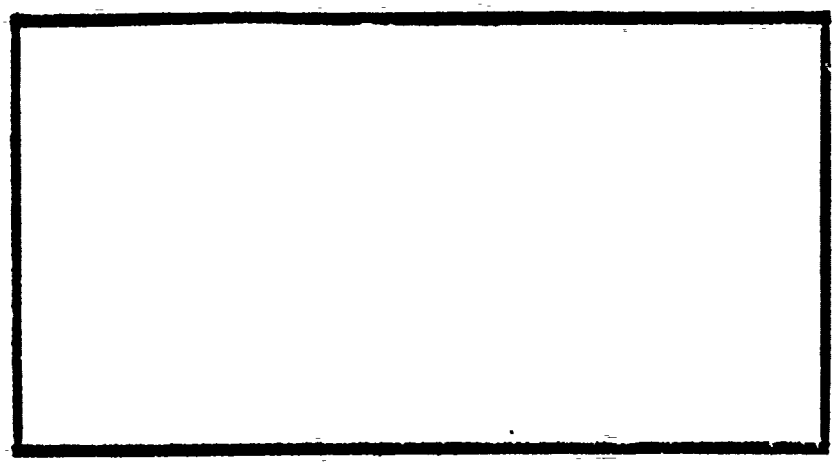




DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GE/ENG/91D-19

A TIME-DEPENDENT ADAPTIVE FILTER FOR COCHANNEL INTERFERENCE REDUCTION

THESIS

Matthew Hunter Foster
Captain, USAF

AFIT/GE/ENG/91D-19

Approved for public release; distribution unlimited

AFIT/GE/ENG/91D-19

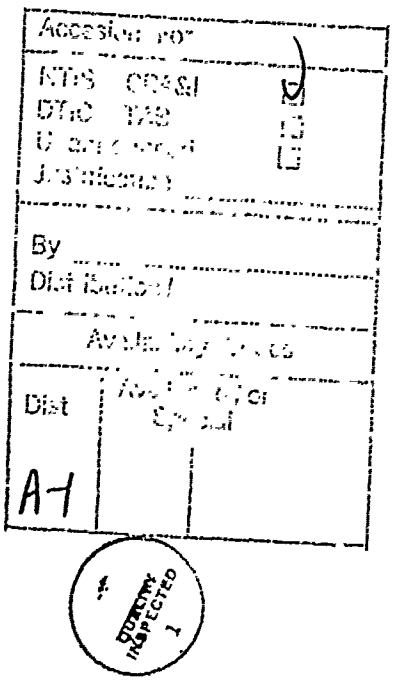
A TIME-DEPENDENT ADAPTIVE FILTER FOR COCHANNEL INTERFERENCE REDUCTION

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Electrical Engineering

Matthew Hunter Foster, B.S.E.E.
Captain, USAF

December, 1991



Acknowledgments

I would like to express my grateful appreciatation to the many people without whose help this thesis would not have been completed. First and formost I must thank my most capable advisor, Mr. Martin P. DeSimio. When things got sticky, Marty always got me unstuck and moving in the right direction. The things that are right about this thesis are thanks to him.

I am grateful also to the other members of my committee, Lt Col David Norman, and Capt Mark Mehalic. Capt Mehalic first introduced me to the subject of Communications at the graduate level, and it was his enthusiasm that sprarked my initial interest in this thesis topic. Col Norman undertook the monumental task of introducing my to Random Signal Theory, and provided me with the tools I needed to complete the theoretical portions of this thesis.

Finally, I am deeply indebted to my wife, Vicki, my daughter Adrienne, and my son Ben, who supported me with love, kind words, and encouragement throughout my stay at AFIT. Had any one of them not given me the strength and support that they did, this thesis would not have come to fruition.

Matthew Hunter Foster

Table of Contents

	Page
Acknowledgments	ii
Table of Contents	iii
List of Figures	vi
List of Tables	viii
Abstract	ix
I. Introduction	1-1
1.1 Background	1-1
1.2 Problem Statement	1-2
1.3 Scope	1-2
1.4 Approach	1-3
1.5 Organization	1-3
II. Background	2-1
2.1 Introduction	2-1
2.2 Cyclostationary Signals	2-1
2.3 Adaptive Filtering	2-5
2.3.1 Some Preliminaries.	2-5
2.3.2 The Least Mean Squared (LMS) Algorithm.	2-8
2.3.3 Time Dependent Adaptive Filtering.	2-11
2.4 Chapter Summary	2-11

Page

III. Simulation Implementation and Verification 3-1

3.1 Introduction 3-1

3.2 Top Level Description of the Simulation 3-1

3.2.1 BER Version and LC Version 3-3

3.2.2 Input, Output, Notation, and Definitions. . . 3-3

3.3 Simulation Details 3-5

3.3.1 Data Generator. 3-5

3.3.2 Finite Impulse Response Low Pass Filters. . . 3-6

3.3.3 Data Formatter 3-7

3.3.4 Modulator and Demodulator 3-8

3.3.5 Noise Generator 3-9

3.3.6 Time Independent Adaptive Filter (TIAF) . . 3-10

3.3.7 Time Dependent Adaptive Filter (TDAF) . . 3-12

3.4 Code Verification 3-18

3.5 Chapter Summary 3-21

IV. Results 4-1

4.1 Introduction 4-1

4.2 Simulation in a Noisy Environment 4-1

4.2.1 Improvement Factor. 4-4

4.2.2 Comparison of Filtered Demodulated Signals. 4-5

4.2.3 Comparison of MSE Learning Curves. 4-5

4.2.4 Bit Error Rate 4-12

4.2.5 Summary 4-12

4.3 Simulation in Interference 4-12

4.3.1 Improvement Factor. 4-13

4.3.2 Comparison of Filtered Demodulated Signals. 4-16

4.3.3 Comparison of MSE Learning Curves. 4-16

Page

4.3.4 Bit Error Rate 4-22

4.3.5 Summary. 4-22

4.4 Varying the Baud Rate of the Interferer 4-23

4.5 Varying the Carrier Frequency of the SNOI 4-23

4.6 Chapter Summary 4-25

V. Conclusions and Recommendations 5-1

5.1 Conclusions 5-1

5.2 Recommendations 5-2

Appendix A. Input Parameters and Output Files for the BER Version A-1

Appendix B. Input Parameters and Output Files for the LC Version B-1

Appendix C. Source Code for the BER Version C-1

Appendix D. Source Code for the LC Version D-1

Bibliography BIB-1

Vita VITA-1

List of Figures

Figure — Page

2.1. A quadratic mean squared error surface. The verticle axis is ξ and the horizontal axes are the filter weights 2-9

3.1. Signal flow through the simulation 3-2

3.2. An Adaptive Linear Combiner 3-11

3.3. Pseudocode for the TIAF algorithm 3-13

3.4. A TDAF implemented as a parallel bank of TIAFs 3-14

3.5. The impulse train described by Eq 3.12 sampled every T_s 3-15

3.6. Stationarization of a periodic pulse train by a TDAF 3-17

3.7. Pseudocode for the TDAF algorithm 3-19

3.8. Convergence of the adaptive filters with no interference or noise: a) the desired signal; b) the TIAF response; c) the TDAF response. 3-20

4.1. Improvement factor as a function of SNR 4-4

4.2. The desired signal, TDAF response, and TIAF response for 0 dB SNR 4-6

4.3. The desired signal, TDAF response, and TIAF response for -5 dB SNR 4-6

4.4. The desired signal, TDAF response, and TIAF response for -10 dB SNR 4-7

4.5. The desired signal, TDAF response, and TIAF response for -15 dB SNR 4-7

4.6. The desired signal, TDAF response, and TIAF response for -20 dB SNR 4-8

4.7. MSE learning curves for the TDAF and TIAF for 0 dB SNR . . . 4-9

4.8. MSE learning curves for the TDAF and TIAF for -5 dB SNR . . 4-10

Figure Page

4.9. MSE learning curves for the TDAF and TIAF for -10 dB SNR . . 4-10

4.10. MSE learning curves for the TDAF and TIAF for -15 dB SNR . . 4-11

4.11. MSE learning curves for the TDAF and TIAF for -20 dB SNR . . 4-11

4.12. Improvement factor as a function of SIR 4-16

4.13. The desired signal, TDAF response, and TIAF response for 0 dB SIR 4-17

4.14. The desired signal, TDAF response, and TIAF response for -5 dB SIR 4-17

4.15. The desired signal, TDAF response, and TIAF response for -10 dB SIR 4-18

4.16. The desired signal, TDAF response, and TIAF response for -15 dB SIR 4-18

4.17. The desired signal, TDAF response, and TIAF response for -20 dB SIR 4-19

4.18. The SNOI present in Figures 4.13 through 4.17 4-19

4.19. MSE learning curves for the TDAF and TIAF for 0 dB SIR . . . 4-20

4.20. MSE learning curves for the TDAF and TIAF for -5 dB SIR . . . 4-20

4.21. MSE learning curves for the TDAF and TIAF for -10 dB SIR . . 4-21

4.22. MSE learning curves for the TDAF and TIAF for -15 dB SIR . . 4-21

4.23. MSE learning curves for the TDAF and TIAF for -20 dB SIR . . 4-22

4.24. MSE as a function relative baud rate at various SIRs for the TDAF 4-24

4.25. MSE as a function relative baud rate at various SIRs for the TIAF 4-24

4.26. MSE as a function SNOI carrier frequency. SOI carrier frequency fixed at 6 Hz 4-25

List of Tables

Table Page

4.1. SOI carrier amplitude and noise gain for noisy environment simulations 4-2

4.2. Input parameters for noisy environment simulations for the BER version 4-2

4.3. Input parameters for noisy environment simulations for the LC version 4-3

4.4. MSE data for simulations in AWGN 4-5

4.5. BER for five simulations in a noisy environment 4-12

4.6. SOI and SNOI carrier amplitude for interference environment simulations 4-13

4.7. Input parameters for interference environment simulations for the BER version 4-14

4.8. Input parameters for interference environment simulations for the LC version 4-15

4.9. MSE data for simulations in interference 4-15

4.10. BER for five Simulations in Interference 4-22

Abstract

This thesis presents a Time Dependent Adaptive Filter (TDAF) which exploits the cyclostationarity of digitally modulated communications signals and seeks to improve the Signal to Interference Ratio (SIR) and Signal to Noise Ratio (SNR) of such signals. The TDAF is imbedded in a computer simulation of a simple communication system consisting of a data source, data formatter, pulse shaping filter, BPSK modulator, and demodulator. In the simulation the TDAF and a Time Independent Adaptive Filter (TIAF) attempt to extract the Signal of Interest (SOI) from noise or interference. The criteria of Mean Squared Error (MSE) is used as the primary means to compare the performance of the two adaptive filters. Plots of MSE improvement in interference and MSE improvement in noise are presented. For the case of interference, the improvement is measured as a function of the baud rate of the intereference signal, and carrier frequency of the interference signal. It is shown that with respect to the TIAF, the TDAF provides up to 12 dB of improvement. Bit Error Rates (BER) for several simulations are presented. The data indicate that signficant improvements in BER might be expected when a TDAF is used in lieu of a TIAF.

A TIME-DEPENDENT ADAPTIVE FILTER FOR COCHANNEL INTERFERENCE REDUCTION

I. Introduction

1.1 Background

The advent of frequency reuse, particularly in satellite communications, has made the task of recovering digitally modulated signals more challenging. Unoccupied frequencies in the electromagnetic spectrum either do not exist or are impractical for use due to their high frequency or poor channel performance (in the O_2 and H_2O absorption bands, for instance). A single communication system may reuse frequencies up to six times through polarization and spatial (antenna pointing) reuse techniques. INTELSAT VI is an example of such a system (3:99).

The classical technique of applying the signal of interest (SOI) to a relatively narrow passband filter can be of limited effectiveness when the SOI has been corrupted by another signal (or signals) not of interest (SNOI) whose spectral components overlap those of the SOI. Of course, with only knowledge of the center frequency and spectral width of the SOI, a digital filter can be readily designed that limits power outside the spectral region of interest (5:403-489). The impulse response of such a filter is fixed. Hence, if the characteristics of the SNOI changes, the filter cannot appropriately adapt to the new conditions.

If, on the other hand, the filter coefficients are allowed to vary so as to minimize some error criteria, the impulse response will no longer be fixed. Then, even if the characteristics of the SNOI change, the filter can adapt to a new 'optimum' solution. Such a filter is referred to as a 'Time Independent Adaptive Filter' (TIAF); 'adaptive' because the impulse response changes according to the characteristics of the input, and 'time independent' because the changes in the impulse response are not an explicit function of time, but rather a function only of the input to the filter. An example of a TIAF is the adaptive linear combiner (ALC) (10:15-26).

The error in a TIAF is a quadratic function of the filter weights. The quadratic surface resembles a bowl (10:19, 20). The bottom of the bowl represents the minimum error, and the filter strives to achieve that operating point. When the statistics of the input are stationary or nearly stationary (change very slowly), the filter will be able to achieve and maintain a nearly optimum configuration (11). Unfortunately, all digitally modulated waveforms of interest in modern communication theory are not stationary. They exhibit some periodicity in their statistics referred to as cyclostationarity (2:16-18). The TIAF is in general not able to adapt quickly enough to 'track' the optimum solution that exists for a cyclostationary SOI.

What is needed is some way to make the input to the filter stationary so that it may better track the optimum solution. One way to do this when filtering samples from a digitally modulated signal is to implement the filter by using multiple TIAFs arranged in parallel, and commutating the output from each of the TIAFs. Then, each TIAF has its own optimum solution, or 'bowl', and the filter can acheive significantly reduced overall MSE (1:681). The selection of the proper TIAF is an explicit function of time, and hence, an adaptive filter so constructed is called a 'time dependent adaptive filter' (TDAF).

1.2 Problem Statement

This thesis presents a TDAF which can be used to improve the signal to interference ratio (SIR) and signal to noise ratio (SNR) of digitally modulated communications signals. The performance improvement of the TDAF over the TIAF is determined based on the application of various metrics, including Mean Square Error (MSE) and Bit Error Rate (BER).

1.3 Scope

The intent of this thesis is to produce a computer simulation of a simple digital communications system that can be used to evaluate the performance of a TDAF under realistic conditions. A binary phase shift keyed (BPSK) signal is used as both the SOI and the SNOI. Comparisons of performance are made between the TDAF and the TIAF for varying SNR, SIR, interference carrier frequency, and interference data rate.

1.4 Approach

The approach used to complete this research is divided into three phases. The first phase has two parts. The first part is basic research into adaptive filters and cyclostationary signals. The second part is the development of the framework for the computer simulation of the communication system. The second phase is the development of the TDAF itself, and its insertion into the simulation. The last phase is the characterization of the TDAF. Measurement of the TDAFs ability to accurately recover the signal is made by running simulations for various interference characterizations.

1.5 Organization

Chapter II provides a discussion of the fundamental concepts of adaptive filters and cyclostationarity. Chapter III contains details on the construction of the TDAF simulation. Chapter IV presents the results and analysis of data generated by the environment. The fifth and final chapter contains specific conclusions along with recommendations for future research.

II. Background

2.1 Introduction

In order to understand how Time Dependent Adaptive Filters (TDAF) improve the Signal to Noise Ratio (SNR) and Signal to Interference Ratio (SIR) of digitally modulated bandpass signals, there are several concepts that need to be understood. Among these are the concepts of cyclostationarity and adaptive filtering. In presenting the idea of cyclostationarity, this chapter will provide the definition of cyclostationarity, and give an example of a cyclostationary signal. Next, a brief introduction to adaptive filtering will be presented. The Adaptive Linear Combiner (ALC) which is a type of Time Independent Adaptive Filter (TIAF) will be presented. The TIAF is presented first because the TDAF can be implemented by combining multiple TIAFs in parallel. Finally, the concept of Time Dependent Adaptive Filtering will be briefly introduced. A thorough description of the implementation of the TDAF is reserved for Chapter III.

2.2 Cyclostationary Signals

A process is cyclostationary if it has a periodic components in its autocorrelation function (2:20). The Fourier series expansion of a periodic (with period T_0) function $x(t)$ is

$$x(t) = \sum_{n=-\infty}^{\infty} C_n e^{j2\pi\alpha t} \tag{2.1}$$

where

$$\alpha = \frac{n}{T_0}$$

where the coefficient at any given frequency is

$$C_n = \frac{1}{T_0}\int_{T_0} x(t)e^{-j2\pi\alpha t}dt \tag{2.2}$$

$$= \langle x(t)e^{-j2\pi\alpha nt}\rangle \tag{2.3}$$

Recall that the power spectral density (PSD) of any function is the Fourier transform of its autocorrelation. Therefore, if it can be shown that the PSD of a function exhibits delta functions, then the function is cyclostationary.

For ergodic functions, the classical autocorrelation function is given by

$$R_{xx}(\tau) = \langle x(t)x(t+\tau)\rangle \tag{2.4}$$

where

$$\begin{aligned} x(t) &= \text{the function of interest} \\ \tau &= \text{the lag value} \end{aligned}$$

Notice that it is mathematically equivalent to write Eq 2.4 as

$$R_{xx}(\tau) = \langle x(t+\tau/2)x(t-\tau/2)\rangle \tag{2.5}$$

even though Eq 2.5 is not physically realizable (because of its noncausal nature).

The cyclic correlation function is a straightforward extension of this definition (2:19)

$$R_x^\alpha(\tau) = \langle x(t)x(t+\tau)e^{-j2\pi\alpha t}\rangle \tag{2.6}$$

where

$$\begin{aligned} x(t) &= \text{the function of interest} \\ \tau &= \text{the lag value} \\ \alpha &= \text{the cycle frequency} \end{aligned}$$

There are two equivalent interpretations of Eq 2.6. First, the cyclic autocorrelation function is the standard autocorrelation but with a time *and* frequency shifted version of itself. Referring to Eq 2.2, it can be seen that a second interpretation is that the cyclic autocorrelation is the Fourier coefficient of the standard autocorrelation at a given cycle frequency, α (2:17- 20). Notice that when $\alpha = 0$, Eq 2.6 reduces to the standard autocorrelation function (2:20).

Wide Sense Stationary (WSS) random processes can be either cyclostationary or purely stationary, but not both. Purely stationary processes are those for which no $\alpha \neq 0$ can be found to satisfy $R_x^\alpha(\tau) \neq 0$ except perhaps for the degenerate case where $\tau = 0$ (2:20).

If Eq 2.6 is re-expressed in the form of Eq 2.5, and $e^{-j2\pi\alpha t}$ is factored into $e^{-j\pi\alpha(t+\tau/2)}$ and $e^{-j\pi\alpha(t-\tau/2)}$, then a third interpretation becomes apparent. The cyclic autocorrelation function can be written as a conventional crosscorrelation (2:19)

$$\begin{aligned} R_x^\alpha(\tau) &= \langle [x(t+\tau/2)e^{-j\pi\alpha(t+\tau/2)}][x(t-\tau/2)e^{-j\pi\alpha(t-\tau/2)}] \rangle \\ &= \langle u(t+\tau/2)v(t-\tau/2) \rangle \qquad (2.7) \\ &= R_{uv}(\tau) \qquad (2.8) \end{aligned}$$

where

$$\begin{aligned} u(t) &= x(t)e^{-j\pi\alpha t} \\ v(t) &= x(t)e^{+j\pi\alpha t} \end{aligned}$$

Assuming that at least one of $u(t)$ or $v(t)$ is a zero mean process, the conventional cross-correlation coefficient is (8:124)

$$\gamma_{uv}(\tau) = \frac{R_{uv}(\tau)}{\sqrt{R_u(\tau)R_v(\tau)}} \qquad (2.9)$$

A normalization factor for the cyclic autocorrelation follows from Eq 2.9 and is called the temporal correlation coefficient (2:20)

$$\gamma_x^\alpha(\tau) \triangleq \frac{R_x^\alpha(\tau)}{R_x(0)} \qquad (2.10)$$

The following example was taken from (2) and is summarized here to show how to calculate a cyclic autocorrelation.

Given a real random purely stationary signal with zero mean, $a(t)$ we can write

$$\langle a(t) \rangle = 0 \qquad (2.11)$$

We require that the autocorrelation of $a(t)$ be nonzero:

$$\langle a(t+\tau/2)a(t-\tau/2) \rangle \not\equiv 0 \qquad (2.12)$$

Since we defined $a(t)$ to be purely stationary (rather than cyclostationary), we know that

$$\langle a(t+\tau/2)a(t-\tau/2)e^{-j2\pi\alpha t}\rangle \equiv 0 \text{ for all } \alpha \neq 0 \tag{2.13}$$

Eq 2.13 guarantees that

$$\langle a(t)e^{-j2\pi\alpha t}\rangle \equiv 0 \text{ for all } \alpha \neq 0 \tag{2.14}$$

Now consider the amplitude-modulated sinewave

$$\begin{aligned} x(t) &= a(t)\cos(2\pi f_0 t + \theta) && (2.15)\\ &= \frac{1}{2}a(t)[e^{j(2\pi f_0 t+\theta)} + e^{-j(2\pi f_0 t+\theta)}] && (2.16)\end{aligned}$$

Multiplying $a(t)$ by $\cos(2\pi f_0 t + \theta)$ simply shifts the spectrum of $a(t)$ to $\pm f_0$ (and reduces its magnitude). Therefore, since Eq 2.14 guarantees that $a(t)$ contains no finite-strength additive sinusoidal components, it is clear that $x(t)$ contains no finite-strength additive sinusoidal components either. In other words, there are not delta functions in its power spectral density function. If we apply the non-linear transformation that is inside the time averaging operator of Eq 2.5 to $x(t)$ we get

$$\begin{aligned} y_\tau(t) &= x(t+\tau/2)x(t-\tau/2)\\ &= a(t+\tau/2)a(t-\tau/2)\frac{1}{4}[e^{j2\pi f_0\tau} + e^{-j2\pi f_0\tau}\\ &\quad + e^{j(4\pi f_0 t+2\theta)} + e^{-j(4\pi f_0 t+2\theta)}] \end{aligned} \tag{2.17}$$

Next, we apply the definition of the cyclic autocorrelation function to $y_\tau(t)$:

$$\begin{aligned} \langle y_\tau e^{-j2\pi\alpha t}\rangle &= \frac{1}{4}e^{j2\pi f_0\tau}\langle a(t+\tau/2)a(t-\tau/2)e^{-j2\pi\alpha t}\rangle\\ &\quad + \frac{1}{4}e^{-j2\pi f_0\tau}\langle a(t+\tau/2)a(t-\tau/2)e^{-j2\pi\alpha t}\rangle\\ &\quad + \frac{1}{4}e^{j2\theta}\langle a(t+\tau/2)a(t-\tau/2)e^{-j2\pi(\alpha-2f_0)t}\rangle\\ &\quad + \frac{1}{4}e^{-j2\theta}\langle a(t+\tau/2)a(t-\tau/2)e^{-j2\pi(\alpha+2f_0)t}\rangle \end{aligned} \tag{2.18}$$

Eq 2.13 guarantees that the first two terms of Eq 2.18 are zero. The last two terms of Eq 2.18 can only be nonzero if the exponent inside the time averaging operator is zero. Clearly, that happens when $\alpha = \pm 2f - 0$. Substituting $\alpha = \pm 2f - 0$ into Eq 2.18, and noting that $R_a(\tau) = \langle a(t + \tau/2)a(t - \tau/2)\rangle$ we have the cyclic autocorrelation function function for $x(t)$:

$$R_x^\alpha(\tau) = \begin{cases} \frac{1}{4}e^{\pm j2\theta}R_a(\tau), & \text{for } \alpha = \pm 2f_0 \\ \frac{1}{2}R_a(\tau)\cos(2\pi f_0\tau), & \text{for } \alpha = 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.19}$$

Referring to Eq 2.7, the correlation of the frequency shifted versions of $x(t)$ represented by $u(t)$ and $v(t)$ should not be surprising. After all, $x(t)$ was formed by frequency shifting $a(t)$ by f_0, and the two spectral images of $x(t)$ are separated by $2f_0$.

2.3 Adaptive Filtering

2.3.1 Some Preliminaries. Given a real WSS random process $X(t)$, we know that the autocorrelation of $X(t)$ can be written as (8:143)

$$R_{XX}(\tau) = E\{X(t)X(t + \tau)\} \tag{2.20}$$

where

$$\begin{aligned} E\{\cdot\} &= \text{the statistical expectation operator} \\ \tau &= \text{the lag value} \end{aligned}$$

If we assume $X(t)$ is an ergodic process, then we can restrict our attention to a single member function of $X(t)$, $x(t)$ (8:178). Furthermore, we can sample $x(t)$ starting at some arbitrary point k so that we are left with a discrete sequence $x[k]$. For N

samples, this sequence can be represented as a column vector $\mathbf{X}_k$

$$\mathbf{X}_k = \begin{bmatrix} x_k \\ x_{k+1} \\ \vdots \\ x_{k+(N-1)} \end{bmatrix} \tag{2.21}$$

forming an N element array. Now the value of the autocorrelation at a specific lag value can be calculated as (11)

$$\phi_{xx}[n] = \frac{1}{N} \sum_{k=0}^{N-1} x[k]x[k+n] \tag{2.22}$$

or equivalently in vector notation

$$\phi_{xx}[n] = \frac{1}{N} \mathbf{X}_k^T \mathbf{X}_{k+n} \tag{2.23}$$

where

n = the lag value

N = the number of samples in the data vector

k = the seqence number or vector element number

and the superscript T represents the vector transpose.

We can now define the autocorrelation matrix of a sampled sequence

$$\mathbf{R} = \begin{bmatrix} \phi_{xx}[0] & \phi_{xx}[-1] & \cdots & \phi_{xx}[-n] \\ \phi_{xx}[1] & \phi_{xx}[0] & \cdots & \phi_{xx}[-n+1] \\ \vdots & \vdots & \ddots & \vdots \\ \phi_{xx}[n] & \phi_{xx}[n-1] & \cdots & \phi_{xx}[0] \end{bmatrix} \tag{2.24}$$

Now let us assume that we are designing a digital finite impulse response (FIR) filter. Given the input data vector $\mathbf{X}_k$, we decide we want the filter output to be the sequence $d[k]$, called the desired response. Then we can define a new vector $\mathbf{P}$, the cross correlation between the desired response and the input vector (10:20)

$$\mathbf{P} = E\{d[k]\mathbf{X}_k\} = \begin{bmatrix} d[k]\mathbf{X}_k \\ d[k]\mathbf{X}_{k-1} \\ \vdots \\ d[k]\mathbf{X}_{k-(N-1)} \end{bmatrix} \tag{2.25}$$

It is now possible to apply the Wiener-Hopf equation to obtain the optimum weight vector (called the Wiener weight vector) (10:22)

$$\mathbf{W}^* = \mathbf{R}^{-1}\mathbf{P} \tag{2.26}$$

The output of an FIR digital filter y_k is just the convolution of the input with the filter weights. This can be expressed in sampled sequence notation (5:21)

$$y[k] = \sum_{n=0}^{N-1} w[n]x[k-n] \tag{2.27}$$

or equivalently in matrix notation (10:17)

$$y_k = \mathbf{W}_k^T\mathbf{X}_k \tag{2.28}$$

Unfortunately, neither $\mathbf{R}$ nor $\mathbf{P}$ will in general be known. In an actual communication system, $x[k]$ will be given by

$$x[k] = d[k] + i[k] + n[k] \tag{2.29}$$

where

$$\begin{aligned} d[k] &= \text{the signal of interest} \\ i[k] &= \text{some interfering signal} \\ n[k] &= \text{noise} \end{aligned}$$

Substituting into Eq 2.22

$$\phi_{xx}[n] = \frac{1}{N}\sum_{k=0}^{N-1}\{d[k]+i[k]+n[k]\}\{d[k-n]+i[k-n]+n[k-n]\} \tag{2.30}$$

In the case of $d[k]$, a special sequence may be transmitted and anticipated by the receiver, and thereby be known *a priori* at the receiver (7:102), but $i[k]$ and $n[k]$ in any practical system are not known. Therefore it is not possible to directly calculate $\mathbf{W}^*$.

2.3.2 The Least Mean Squared (LMS) Algorithm. If the input to the filter is stationary, and $\mathbf{W}$ is allowed to vary according to some rule which tends to minimize the difference between the actual filter response and the desired response, then $\mathbf{W}$ becomes an arbitrarily close approximation of $\mathbf{W}^*$ (11). The equation for the difference between $d[k]$ and the filter response $y[k]$ is simply

$$\epsilon[k] = d[k] - y[k] \tag{2.31}$$

Note that the power in $\epsilon[k]$ is the Mean Squared Error (MSE) and is given by

$$\xi = E\{\epsilon^2[k]\} \tag{2.32}$$

It can be shown (10:19,20) that ξ is a quadratic function of the filter tap weights, the desired response, and the filter input. ξ is referred to as the performance surface, and for a two tap filter, it is a paraboloid (a hyperparaboloid if there are more

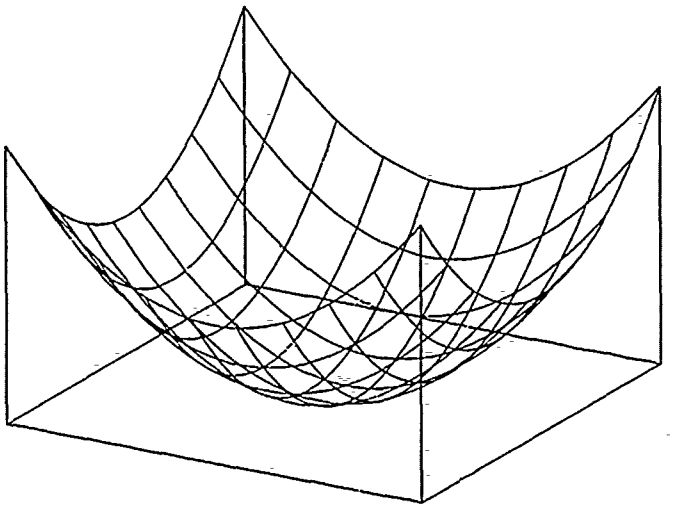


Figure 2.1. A quadratic mean squared error surface. The verticle axis is ξ and the horizontal axes are the filter weights

than 2 weights) and resembles a bowl (see Figure 2.1). The bottom of the bowl is the minimum MSE and when the bowl is projected onto the weight-vector plane, the minimum describes the point in weight space described by $\mathbf{W}^*$ (10:21).

Given that $d[k]$ is known, it is possible to search the error surface for its minimum, thereby arriving at the optimum weight vector $\mathbf{W}^*$. The most straightforward method of searching the error surface is the LMS algorithm (7:101). Other algorithms exist and are presented in (10:Chap 8).

A complete derivation of the LMS algorithm is given in *Adaptive Signal Processing* by Widrow and Stearns (10:99-101); only the result is given here:

$$\mathbf{W}_{k+1} = \mathbf{W}_k + 2\mu\epsilon_k\mathbf{X}_k \tag{2.33}$$

where

$$\mu \;=\; \text{the adaptation coefficient}$$

In words, the weight vector for the next input vector is the sum of the current weight vector and a scaled product of the error and the current input vector.

Eqs 2.27, 2.31, and 2.33 define an adaptive filter incorporating the LMS algorithm. The two important assumptions here are first, that an acceptable estimate of the desired signal d_k is available, and second that $\mathbf{X}_k$ is a stationary process. Should $\mathbf{X}_k$ fail to be a stationary process, then the bottom of the bowl defined by the the surface ξ will exist at different points in weight space. The statistics of $\mathbf{X}_k$ must be non-varying or vary slowly, so that $\mathbf{W}$ will in general approach a near optimum solution (11).

Extending the bowl analogy, searching ξ for its minimum is equivalent to rolling a marble down the side of the bowl. The steepness of the sides of the bowl is determined by the power in $d[k]$ relative to the power in $n[k]$ and $i[k]$ (11). The more power in $d[k]$, the steeper the sides of the bowl; hence, the more rapid the convergence.

With adaptive filters, given an error surface ξ, there is a trade-off between rapid convergence and close approximation of $\mathbf{W}^*$. The trade-off involves selection of an appropriate adaptation coefficient, μ. The smaller μ, the more closely $\mathbf{W}$ approaches $\mathbf{W}^*$ (assuming stationarity of the input). For faster adaptation, μ is chosen to be larger, but the MSE also increases. A good rule of thumb is to select μ such that (10:103,111- 114)

$$\mu = \frac{M}{(N+1)(\text{Power in } \mathbf{X}_k)} \tag{2.34}$$

where M is the *misadjustment* of the filter. The misadjustment is a measure of the average distance between $\mathbf{W}$ and $\mathbf{W}^*$ that the filter designer is willing to live with. Smaller M results in close approximation of $\mathbf{W}^*$, but slower adaptation. Eq 2.34

also assures convergence of the filter for $0 < M < 1$. Unlike nonadaptive FIRs, the adaptive transversal filter can fail to converge (10:102).

2.3.3 Time Dependent Adaptive Filtering. Digitally modulated signals are not stationary, but rather cyclostationary (4:1). As a result, there does not exist a single value of $\mathbf{W}^*$ associated with an single error surface ξ. Instead, there exist (for a sampled signal) a finite number of error surfaces (1:679, 680). A TDAF simply provides a separate TIAF for each error surface ξ_j in weight space. This can result in a significant reduction in the MSE of the filter output (7:3). Each TIAF in the TDAF has an independently adapted weight vector, $\mathbf{W}_{k,j}$. The TDAF LMS algorithm is (1:681)

$$y_k = \mathbf{X}_k^T \mathbf{W}_{j,k} \tag{2.35}$$

and

$$\mathbf{W}_{j,k+1} = \begin{cases} \mathbf{W}_{j,k} + 2\mu\epsilon_k \mathbf{X}_k & \epsilon_k \text{ due to } \xi_j \\ \mathbf{W}_{j,k} & \text{otherwise} \end{cases} \tag{2.36}$$

For error surface ξ_j, the weight vector $\mathbf{W}_{k,j}$ is convolved with the input and is subsequently updated according to the LMS algorithm. The other weight vectors $\mathbf{W}_{k,l\neq j}$ are dormant. Hence, the overall time to adaptation for the TDAF is slower than for the TIAF since each weight vector is updated only periodically, rather than at each sample time k as in the case of the TIAF. The specific implementation of the TDAF will be covered in detail in Chapter III.

2.4 Chapter Summary

This chapter provided an overview of the concepts of cyclostationarity and adaptive filtering. A mathematical means by which to determine if a signal is cyclostationary was provided, and an example of the calculation was presented. Time Independent Adaptive Filtering was covered to form a foundation for the introduction of the concept of Time Dependent Adaptive Filtering.

III. Simulation Implementation and Verification

3.1 Introduction

This chapter provides a detailed description of the methodology employed to characterize the performance of the Time Dependent Adaptive Filter designed in support of this research effort. Included is a description of the computer program that was designed and written to facilitate the characterization.

3.2 Top Level Description of the Simulation

The simulation is a computer program that was written in the C programming language, and compliled on the public-domain GNU C compiler. Pains were taken to adhere to the ANSI standard that was recently established for the C language. As a result, the code should compile and run on any system that has an ANSI C compiler and sufficient memory resources, including an IBM AT class compatible computer. In fact, much of the code was developed on a PC compatible computer with an 80386 microprocessor and 80387 floating point coprocessor using Borland C++ in ANSI mode.

The program simulates a simple communication system. Included in the system are a data source, bandlimiting filters, a modulator, a noisy channel, adaptive filters and a demodulator (See Figure 3.1). The SOI channel and SNOI channel shown in Figure 3.1 are identical. The data rate for the SNOI channel can be set to some percentage of the SOI data rate.

In the simulation, the symbol rate of the signal of interest (SOI) channel is 1 symbol/second. All other times, rates, and frequencies are based on that value. While an actual practical communication system is unlikely to use such a data rate, it is a simple matter to scale that rate to the appropriate level. This normalization to

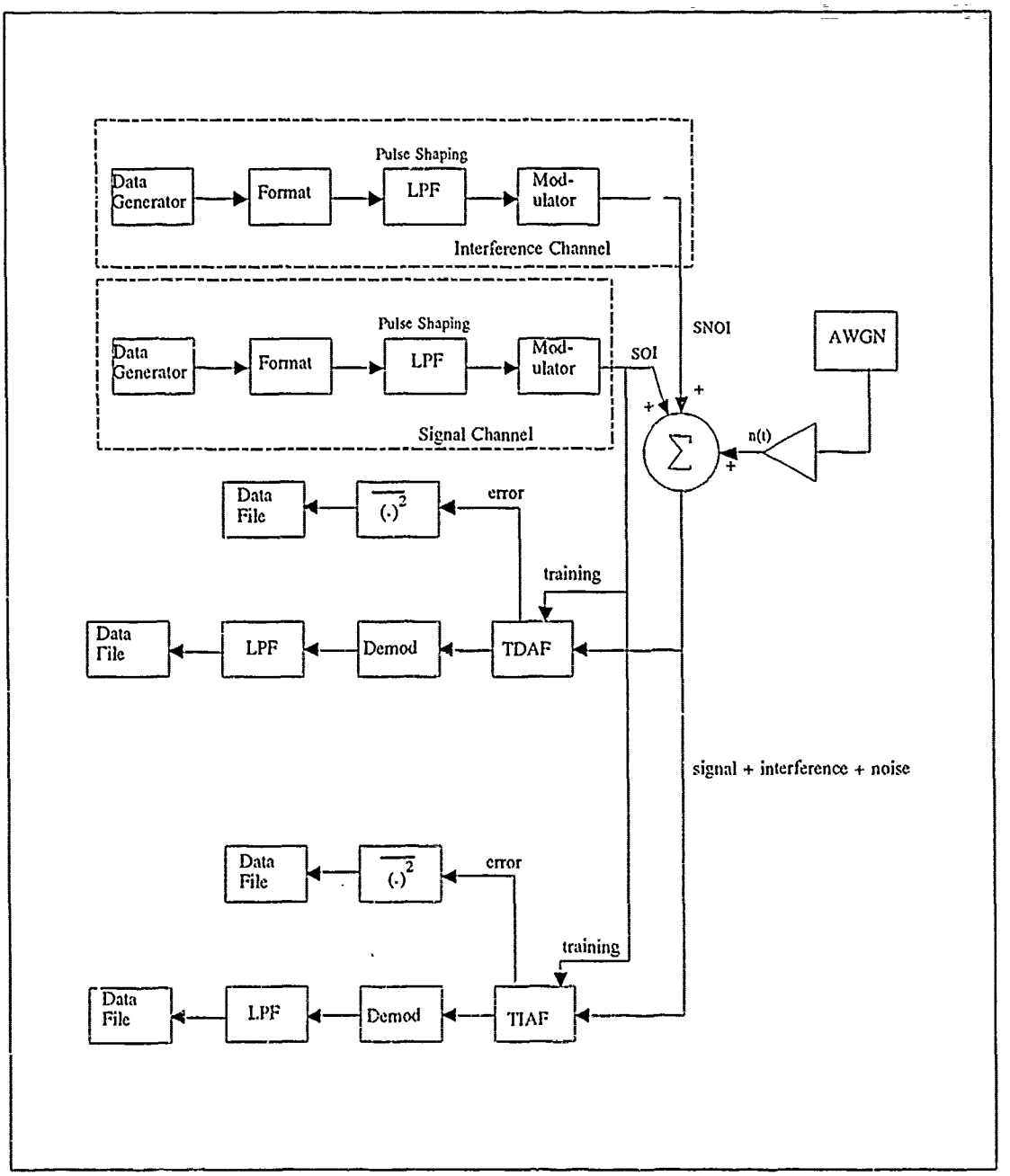


Figure 3.1. Signal flow through the simulation

the SOI data rate simplifies the design of the simulation and analysis of the resulting data.

There is a fundamental "unit of simulation time" referred to as an epoch. An epoch always is comprised of an integer number of symbols. The duration of the simulation is determined by specifying the number of symbols per epoch and the number of epochs in the simulation.

Every reasonable attempt has been made to modularize the software to facilitate modification of the simulation. Each component of the simulation (filters, modulators, etc.) accepts a single value for the input and provides back to the program a single value as output. Hence, it is straightforward to substitute a different filter or modulator for the one provided.

3.2.1 BER Version and LC Version There are two distinct versions of the simulation which differ in the data that they produce, but not in the filtering of the composite channel signal. The first version allows the filters to adapt for a specified number of epochs, and then freezes the weights and continues to run, keeping track of the number of bit errors for the TDAF channel and for the TIAF channel. This is referred to as the BER (Bit Error Rate) Version.

The second version allows the filters to adapt continuously while storing the resulting squared error. After a full epoch, the filter weights are reinitialized and again allowed to adapt. The resulting MSE . averaged with that of the previous epoch. The process is repeated a specified number of epochs (selectable at run time) providing an estimate of the expected value of the MSE as a function of adaptation time. This is called the LC (Learning Curve) Version.

3.2.2 Input, Output, Notation, and Definitions. Depending on the version being run, the simulation requires 24 input parameters at run time. The required parameters can be keyed in at the terminal, but ease of use is significantly enhanced

if the parameters are typed into a text file, and provided to the simulation using the UNIX or DOS redirection operator. The parameters required for the BER version are listed in Appendix A along with a list of the data files produced. The same information for the LC version is listed in Appendix B.

The MSE returned by the BER version of the program is different from the MSE returned by the LC version. The BER version of the program freezes the filter weights, and then starts to collect squared error data. That data is then averaged and returned by the program as the MSE. The LC version allows the filter weights to continuously adapt, and averages squared error sample by sample for all epochs yielding a vector of MSE as a function of sample number. The MSE error for the last P symbols of the learning curve is averaged, and that number is returned by the program as the MSE. As a result, for equal inputs and initialization, the MSE returned by the LC version of the program will always be smaller than that returned by the BER version.

All points in the signal path of the simulation are necessarily discrete samples. In an actual system, many of these signal would be continuous time. In this chapter, continuous time signals are represented as functions of time: $x(t)$, $d(t)$, etc. Discrete time signals are represented as functions of sample number: $x[k]$ or x_k, $d[k]$ or d_k, etc. Here is a list of most of the signals used in the simulation:

- $b_{SOI}(t)$, $b_{SNOI}(t)$: baseband data signal (1 or 0).
- $m_{SOI}(t)$, $m_{SNOI}(t)$: formatted baseband data signal (bi-polar or bi-phase).
- $m'_{SOI}(t)$, $m'_{SNOI}(t)$: bandlimited baseband data signal.
- $d_{SOI}(t)$, $d_{SNOI}(t)$: modulated signal.
- $n(t)$: Additive white gaussian noise.
- $x(t)$: sum of $d_{SOI}(t)$, $d_{SNOI}(t)$ and $n(t)$.

- $y_{TDAF}(t)$, $y_{TIAF}(t)$: adaptively filtered bandpass signal.
- $y'_{TDAF}(t)$, $y'_{TIAF}(t)$: demodulated signal.
- $\hat{m}_{TDAF}(t)$, $\hat{m}_{TIAF}(t)$: recovered signal.
- $\epsilon_{TDAF}(t)$, $\epsilon_{TIAF}(t)$: difference signal, $d(t) - y(t)$

3.3 Simulation Details

The simulation has two time bases: one for the SOI and another for the SNOI. The simulation operates by calculating the signal value at each point in the signal path and then incrementing time by 1/(sample frequency). The signal value is then recalculated. This process is repeated until the required number of data symbols have been processed.

3.3.1 Data Generator. The data generator (DG) used in this simulation produces a pseudorandom sequence of 1's and 0's sampled at the appropriate frequency. Incorporated in the DG is a routine taken directly from *Numerical Recipes in C* (6:226, 228) called *irbit2* which actually calculates the value of each data bit. Whenever a full bit time has elapsed, the DG calls *irbit2*, requesting a new bit value, either 1 or 0, for the random sequence. The DG continues to output that value for the next full bit time.

The routine *irbit2* requires a primitive polynomial modulo 2, referred to here as the seed, to form the random bits. The polynomial used was

$$x^{18} + x^5 + x^2 + x^1 + x^0. \tag{3.1}$$

In C, the above polynomial is represented as the bit sequence

$$\{100000000000100111\} \tag{3.2}$$

and generates a random sequence with a period of repetition of $2^{18}-1 = 262143$ bits. The longest simulation run was 512 epochs $\times$ 16 symbols per epoch = 8192 symbols. Hence, as far as the simulation was concerned, the data sequence was purely random.

A call to the DG routine requires six input parameters:

- *time:* The current value of the base clock for the signal path under consideration (either SOI or SNOI).
- *datarate:* This is constrained to be 1 for the SOI. For the SNOI, it is calculated by the simulation as 1/SNOI symbol frequency where the SNOI symbol frequency is an input parameter to the program.
- *lastdata:* The value of the last sample returned by DG. This variable is modified by the routine.
- *lasttime:* The time when the current bit started.
- *iseed:* The current value of the seed required by *irbit2*. This variable is modified by the routine.
- *outputflag:* A flag which forces the DG to return a square wave rather than random data when it is set.

3.3.2 Finite Impulse Response Low Pass Filters. FIR filters are used for the required low pass filters (LPF) because the filter coefficients can be quickly and easily calculated so that it is possible to select the cutoff frequency of the filter at run time. The FIR filter coefficients are calculated using the windowed Fourier Transform method with the Hamming window(5:444-452):

$$h_k = \frac{\sin[\omega_c(k-N/2)]}{\pi(k-N/2)} w_k \tag{3.3}$$

$$w_k = \begin{cases} 0.54 - 0.46\cos(2\pi k/N), & 0 \le k \le N \\ 0, & \text{otherwise} \end{cases} \tag{3.4}$$

where

$$\begin{aligned} \omega_c &= \text{the discrete cutoff frequency } (0 \leq \omega_c < \pi) \\ N &= \text{the number of taps in the FIR filter} \end{aligned}$$

The output of the filtering routine is the convolution of the filter coefficents given by Eq (3.3) and the past N input values:

$$m'[k] = \sum_{n=0}^{N-1} h[n]m[k-n] \tag{3.5}$$

The filters are used for two purposes at two different points. First, to band limit the baseband message signal (either SOI or SNOI) at the input to the channel, and second, to reject the double frequency component of the demodulated signal at the output of the mixer in the demodulator (see Section 3.3.4).

A call to the LPF routine requires five input parameters:

- *input:* The value of the signal to be filtered.
- *numdelays:* One less than the number of filter coefficients.
- *weights:* A vector containing the values of the filter coefficients.
- *contents:* A vector containing the past *numdelays* samples of the input.
- *gain:* The output of the filter is simply multiplied by this number.

3.3.3 Data Formatter The data formatter (DF) simply converts the binary digits of the data generator into a bi-polar signal. When its input is a 1, the formatter does nothing. When its input is a 0, the formatter returns -1. The formatter is also capable of formatting the output into a Manchester code. However, no data was taken with the formatter running in this mode.

3.3.4 Modulator and Demodulator BPSK modulators have several advantages that lend themselves to use in a simulation such as this. First, they are simple to code. Second, they are in common use. Finally, the demodulated signal, once low pass filtered, is restored to the shape of the baseband signal. As a result, a BPSK modulator was chosen for this simulation.

The equation that describes the modulation process is (9:130)

$$s(t) = Am'(t)\cos(2\pi f_c t) \tag{3.6}$$

where

$$\begin{aligned} A &= \text{carrier amplitude} \\ f_c &= \text{carrier frequency} \\ m'(t) &= \text{the formatted message} \end{aligned}$$

Note that if $m'(t) = \pm 1$, the power in the modulated signal is (9:16)

$$P = \frac{A^2}{2} \tag{3.7}$$

A call to the modulator routine requires four input parameters:

- *input:* The value of the baseband signal.
- *carrierfreq:* The carrier frequency, f_c.
- *carrierampl:* The carrier amplitude, A.
- *time:* The current value of the base clock for the signal path under consideration.

The demodulator simply restores the bandpass signal to baseband. The equation that describes its operation is

$$\hat{m}(t) = y(t)\cos(2\pi f_c t + \hat{\phi}) \tag{3.8}$$

The output of the demodulator is fed to a low pass filter to reject the double frequency component that results from Eq 3.8.

A call to the demodulator routine requires four input parameters:

- *input:* The value of the bandpass signal.
- *carrierfreq:* The carrier frequency, f_c.
- *phase:* A correction factor provided in the event that there is a significant phase shift through the channel.
- *time:* The current value of the base clock for the signal path under consideration.

3.3.5 Noise Generator The signal n(t) is provided by a routine called *gasdev* taken from *Numerical Recipes in C* (6:210. 211, 217). The routine gasdev returns normally (Gaussian) distributed random samples with zero mean and unit variance. The simulation multiplies the returned sample by a gain factor in order achieve noise power other than 0 dBW. As an example, assume the desired level of noise power is -15 dBW. Converting to absolute power levels

$$-15 \text{ dBW } = 10^{-15/10} = 0.03162 \text{ watts} \tag{3.9}$$

A normally distributed random process X with mean μ and variance σ^2 can be converted into a normal random process Z with zero mean and unit variance by applying

$$Z = \frac{X - \mu}{\sigma} \tag{3.10}$$

Therefore, Z can be converted into X by applying

$$X = \sigma Z + \mu \tag{3.11}$$

Since σ^2 is the ac power in the process (we want to maintain zero mean, hence zero dc), multiplying Z by the square root of the desired power yields the appropriate random process. So in the example,

$$\begin{aligned} \sigma^2 &= 0.03162 \\ \sigma &= 0.17783 \end{aligned}$$

Providing the simulation with a noise gain of 0.17783 at run time results in -15 dBW of noise power being added to the SOI and SNOI.

The only input parameter gasdev requires from the simulation is a seed for the random number generator. Any negative integer on the interval [-65536, -1] is acceptable.

3.3.6 Time Independent Adaptive Filter (TIAF) The TIAF is adapted from Chapter 6 of Widrow and Stearns *Adaptive Signal Processing* (10). The transversal adaptive linear combiner (ALC) has the advantage that it is easy to code and the single input, single output characteristic of the filter is perfectly suited to this simulation. A block diagram of a transversal adaptive linear combiner is shown in Figure 3.2. Note the presence of a bias weight in Figure 3.2. This allows for more rapid convergence of the filter when some dc or very low frequency component is present in the signal being filtered (1:679).

The TIAF routine operates on two vectors which are passed in to the routine each time it is called. The first is the vector $\mathbf{W}_k$ containing the filter coefficients. The second is the input vector $\mathbf{X}_k$ that contains the past N inputs to the filter, where N is the number of filter coefficients.

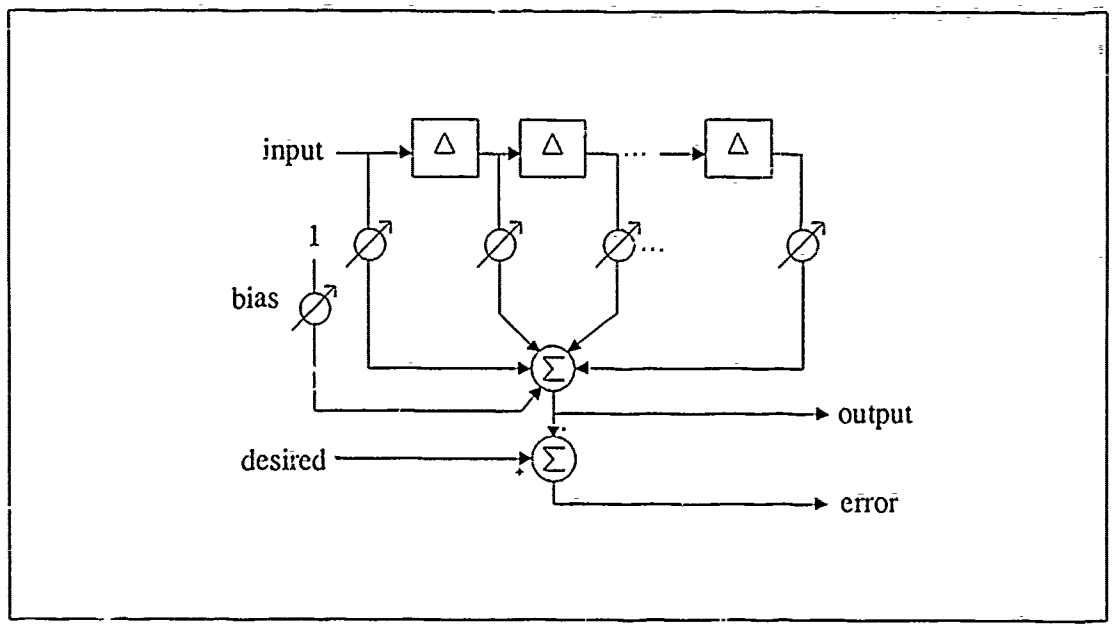


Figure 3.2. An Adaptive Linear Combiner

The TIAF is initialized by calling a routine that sets $\mathbf{W}_k = \mathbf{X}_k = 0$. In the BER version of the simulation, the TIAF is initialized only once. In the LC version, the TIAF is re-initialized after every P symbols, where P is the number of symbols in one epoch.

A call to the TIAF routine requires eight input parameters:

- *input:* The signal being filtered.
- *desired:* The training signal.
- *mu:* The adaptation coefficient.
- *numtaps:* The number of filter coefficients.
- *error:* The value of $d_k - y_k$. Recall that d_k is the desired signal and y_k is the value returned by the routine. This variable is modified by the routine.
- *weights:* The vector $\mathbf{W}_k$ representing the filter weights. This vector is modified by the routine.

- *contents:* The vector $\mathbf{X}_k$ representing the past inputs to the filter. This vector is modified by the routine.
- *bias:* The value of the bias filter weight. This variable is modified by the routine.

When the TIAF routine is called, the process illustrated in Figure 3.3 is performed.

3.3.7 Time Dependent Adaptive Filter (TDAF) The signal $m'(t)$ defined in Section 3.2.2 meets the criteria for $a(t)$ defined by Eqs 2.11, 2.12 and 2.13 inSection 2.2. Therefore the output of the modulator ofSection 3.3.4, $y(t)$, is a cyclostationary signal with cycle frequencies $\pm 2f_c$.

The TDAF is made up of a number of TIAFs with a common input. The output from the TDAF is commutated from the output of the TIAFs as shown in Figure 3.4. The number of TIAFs used is equal to the number of samples taken per symbol of the SOI. Hence, *a priori* knowledge of the sampling rate and symbol rate of the SOI is required to implement this filter. Note that the memory requirements for the TDAF are larger than that for the TIAF, because of the multiple weight vectors required. The processing time is approximately the same for both filters however, since each call to either routine involves only a single convolution.

To show how a TDAF stationarizes a cyclostationary signal, consider the time domain signal consisting of a periodic pulse train with period T_0 and sampling frequency $1/T_s$ (see Figure 3.5):

$$x(t) = \sum_{n=-L}^{L-1} \delta(t - nT_0) \tag{3.12}$$

Let

$$y(t,\tau) = x(t)x(t+\tau) = \begin{cases} x(t), & \tau = kT_0;\ k = 0, \pm 1, \pm 2, \cdots \\ 0, & \text{otherwise} \end{cases} \tag{3.13}$$

```
{Initialize variable that will hold output value...                    }

sum = 0.0;

{An N element vector ranges from x0 to x(N-1)...                       }
{Shift the input vector xk to the right and multiply input by
filter weights while summing the product...                            }

for count = (N - 1) to 1 do
    begin
        xcount = xcount-1;
        sum = sum + x count * w count;
    end;

{now take care of the current input                                    }

x0 = input;
sum = sum + x0 * w0;

{now take care of the bias input                                       }

sum = sum + 1 * wbias;

{The current value of sum will be returned to the program as the
output...                                                              }
{but the error must still be calculated                                }

error = desired - sum;

{Apply the LMS algorithm to the filter weights...                      }

for count = 0 to N - 1 do
    begin
        wcount = wcount + 2 * μ * error * xcount;
    end;
return sum;
```

Figure 3.3. Pseudocode for the TIAF algorithm

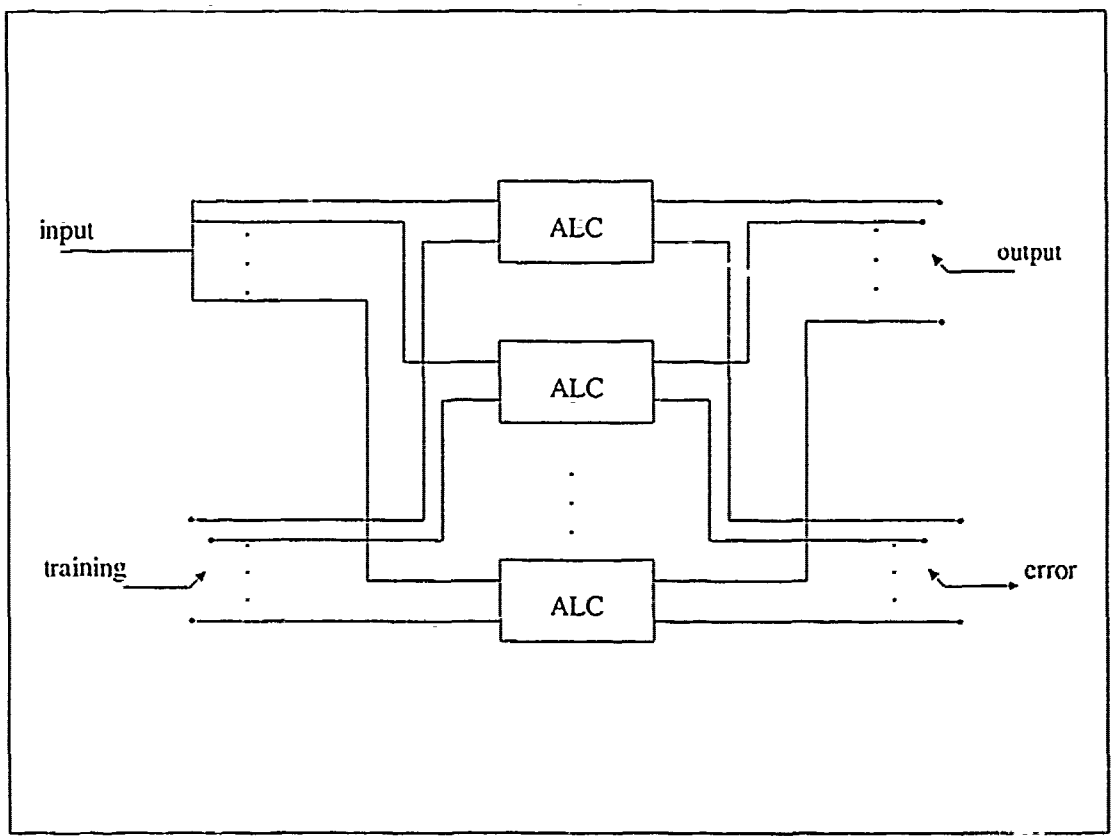


Figure 3.4. A TDAF implemented as a parallel bank of TIAFs

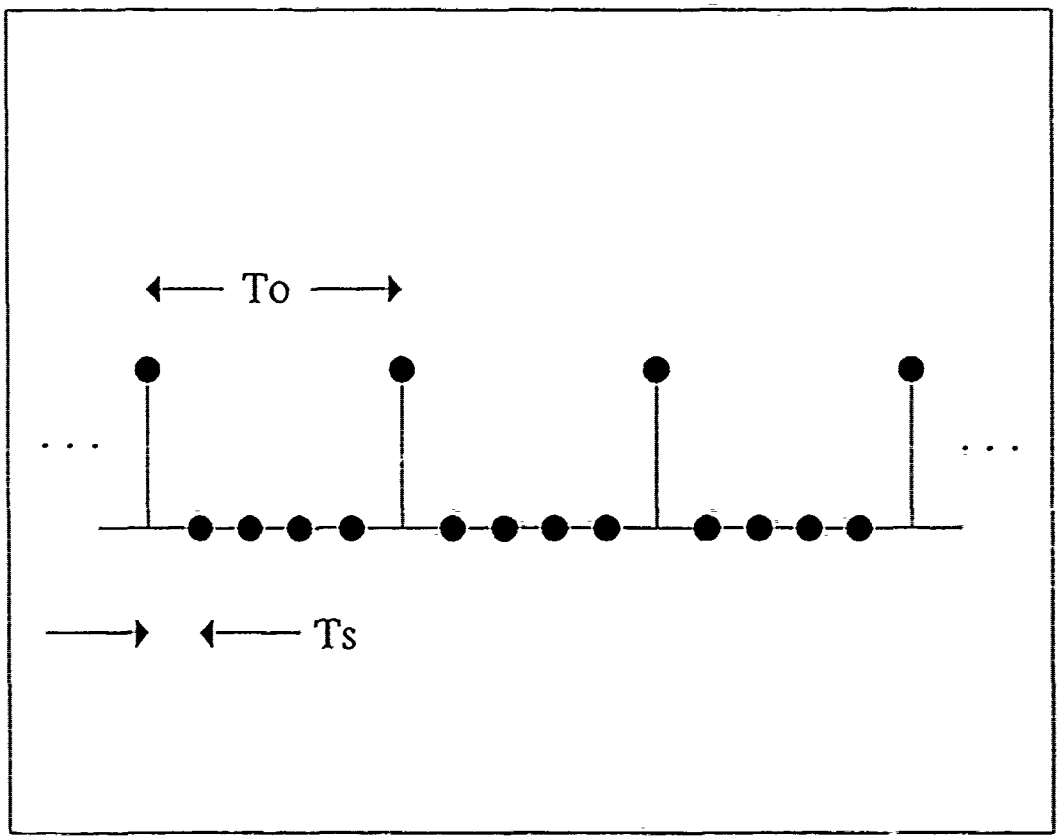


Figure 3.5. The impulse train described by Eq 3.12 sampled every T_s

From Eq 2.6 we have

$$R_x^\alpha(\tau) = \langle y(t,\tau)e^{-j2\pi\alpha t}\rangle \tag{3.14}$$

$$= \lim_{T_0\to\infty}\frac{1}{T_0}\int_{T_0} y(t,\tau)e^{-j2\pi\alpha t}dt \tag{3.15}$$

$$= \lim_{T_0\to\infty}\frac{1}{T_0}\int_{T_0} y(t,\tau)[\cos(2\pi\alpha t) - j\sin(2\pi\alpha t)]\,dt \tag{3.16}$$

$$= \lim_{T_0\to\infty}\frac{1}{T_0}\int_{T_0} y(t,\tau)\cos(2\pi\alpha t)\,dt - \frac{j}{T_0}\int_{T_0} y(t,\tau)\sin(2\pi\alpha t)\,dt \tag{3.17}$$

Recall (Section 2.2) that a signal is cyclostationary if $R_x^\alpha(\tau) \neq 0$ for some $\alpha \neq 0$. Substituting Eq 3.12 into Eq 3.17 and picking $\alpha = 1/T_0$, after interchanging the

order of summation and integration, we have

$$R_x^\alpha(kT_0) = \frac{1}{T_0}\sum_{n=-L}^{L-1}\int_{T_0}\delta(t-nT_0)\cos(2\pi\frac{1}{T_0}t)\,dt - \frac{j}{T_0}\sum_{n=-L}^{L-1}\int_{T_0}\delta(t-nT_0)\sin(2\pi\frac{1}{T_0}t)\,dt \quad (3.18)$$

$$= \frac{1}{T_0}\sum_{n=-L}^{L-1}(\cos(2\pi\frac{1}{T_0}nT_0) - j\sin(2\pi\frac{1}{T_0}nT_0) \quad (3.19)$$

$$= 2L/T_0 \quad (3.20)$$

which is clearly not equal to zero. Therefore Eq 3.12 defines a cyclostationary signal.

Now consider that $x(t)$ is applied to the filter of Figure 3.4. In the first TIAF of the bank the pulse occurs at the first delay element; in the second TIAF, the pulse occurs at the second delay element, and so on. After a full period of the signal, the next pulse in the train will be applied to the filter, where it will occur at the first delay element of the first TIAF. See Figure 3.3.7. Thus, at the time that any given TIAF in the TDAF is updated, the pulse will always be in the same bin. Therefore, from the perspective of the *Nth* TIAF, the signal can be written as the nonperiodic function

$$x_{TIAF}(t) = \delta(t-N) \quad (3.21)$$

The autocorrelation of $x_{TIAF}(t)$ exists *only* for $\tau = 0$. Therefore, except for the degenerate case of $\tau = 0$ the cyclic autocorrelation is zero. Hence, the TIAF sees a purely stationary signal.

A call to the TDAF routine requires nine input parameters:

- *input:* The signal being filtered.
- *desired:* The training signal.
- *mu:* The adaptation coefficient.

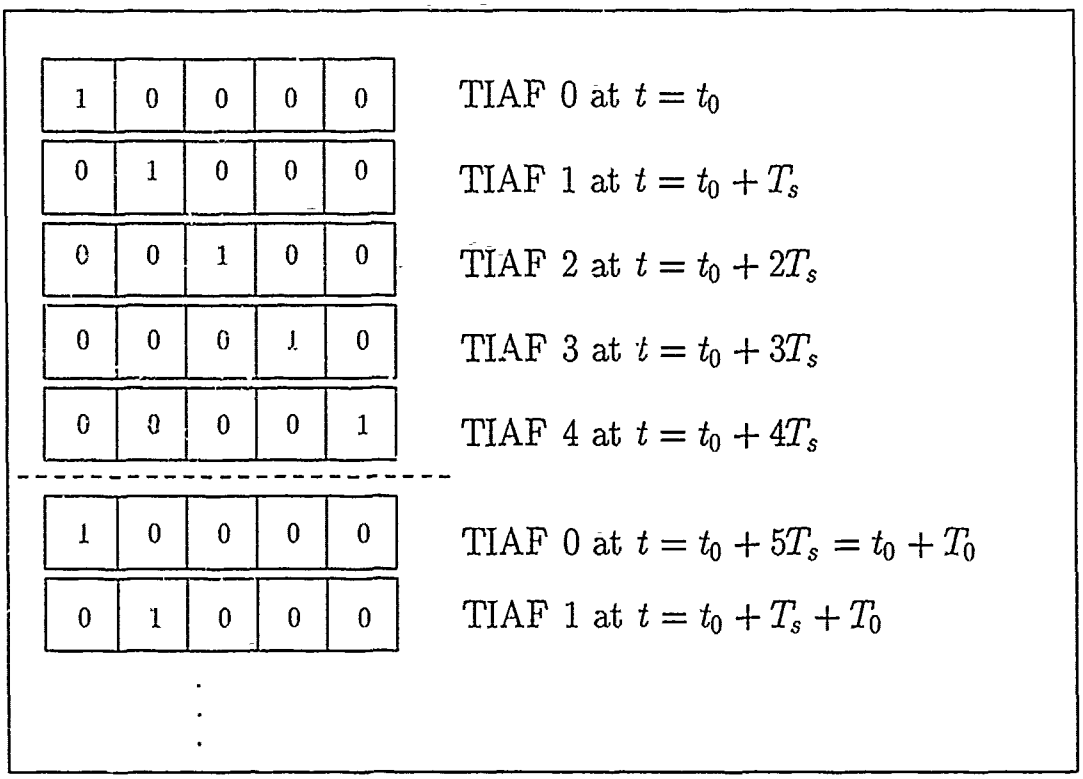


Figure 3.6. Stationarization of a periodic pulse train by a TDAF

- *sampersym:* The number of samples per symbol of $m'(t)$ (also the number of TIAFs in the TDAF).
- *numtaps:* The number of filter coefficients in each TIAF.
- *tdafwts:* The two dimensional array $\mathbf{W}_{j,k}$ representing the filter weights. This array is modified by the routine.
- *contents:* The vector $\mathbf{X}_k$ representing the past inputs to the filter. This vector is modified by the routine.
- *error:* The array containing the values of $d_k - y_k$. Each TIAF has an associated error, hence, the error must be stored in an array. This array is modified by the routine.
- *bias:* The array containing the values of the bias filter weights. Each TIAF has an associated bias weight. This variable is modified by the routine.

Whenever the TDAF routine is called, the process illustrated in Figure 3.7 is performed. Note the similarity to Figure 3.3.

3.4 Code Verification

When properly coded, the adaptive filters in this simulation converge to a zero mean squared solution if there is no interference or noise in the input signal (i.e. when $x(t) = d(t)$) (11). Figure 3.8 shows the converged response of the filters under those circumstances.

The small amount of ripple visible in the response of the TDAF is due to its slow convergence. This was verified by decreasing the adaptation time and noting increased ripple. Each input of the TIAF gives rise to an adjustment of the filter weights. For the TDAF, however, the individual TIAFs comprising the filters are adjusted only once per symbol. Therefore, the TDAF converges much more slowly.

```
{Initialize variable that will hold output value...                    }

sum = 0.0;

{If this is the n^th call to the routine, then the TIAF that is active is...  }

tiafnum = (n mod sampersym) + 1;

{Increment n for the next call to the routine                          }

n = n + 1;

{Shift the input vector x_k to the right and multiply input by
filter weights while summing the product...                            }

for count = (N - 1) to 1 do
    begin
        x_count = x_count-1;
        sum = sum + x count * w count, tiafnum;
    end;

{now take care of the current input                                    }

x_0 = input;
sum = sum + x_0 * w_0, tiafnum;

{now take care of the bias input                                       }

sum = sum + 1 * w_bias, tiafnum;

{The current value of sum will be returned to the program as the
output, but the error must still be calculated...                      }

error tiafnum = desired - sum;

{...and the filter weights must be updated...                          }

for count = 0 to N - 1 do
    begin
        w_count, tiafnum = w_count, tiafnum + 2 * μ * error tiafnum * x_count;
    end;
return sum;
```

Figure 3.7. Pseudocode for the TDAF algorithm

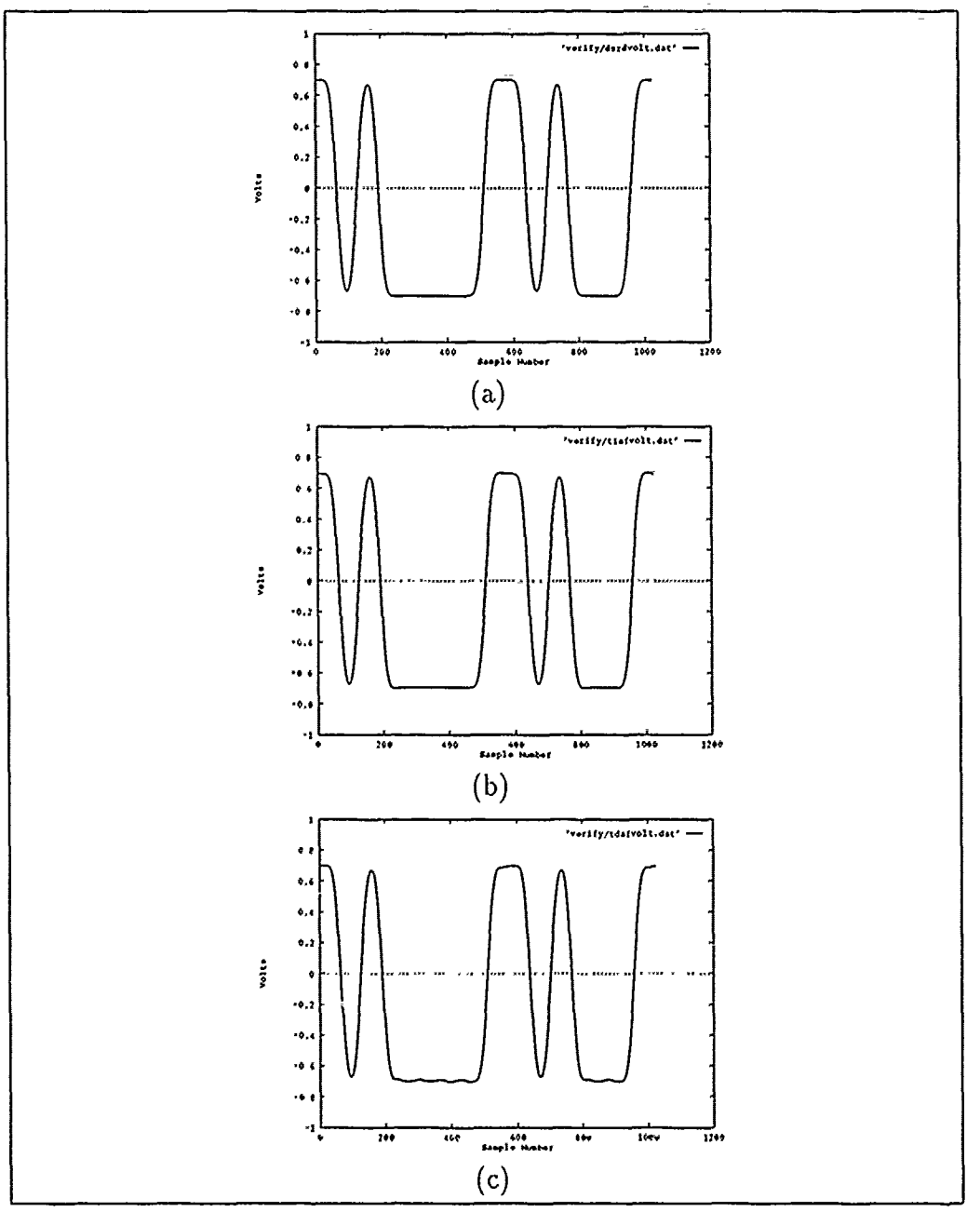


Figure 3.8. Convergence of the adaptive filters with no interference or noise: a) the desired signal; b) the TIAF response; c) the TDAF response.

3.5 Chapter Summary

This chapter provided a detailed description of the simulation developed to compare the performance of a Time Dependent Adaptive Filter with that of a Time Independent Adaptive Filter. In the simulation, both the TDAF and TIAF simultaneously filter the channel signal. The simulation calculates the resultant Mean Squared Error for both filters. another version of the simulation also counts the number of bit errors at the receiver end.

IV. Results

4.1 Introduction

In this chapter, the results obtained from a number of simulation runs are presented. Each simulation run varied one or more of the following parameters:

- Signal to Noise Ratio (SNR) at the filter input
- Signal to Interference Ratio (SIR) at the filter input
- Baud rate of interferer
- Carrier frequency of the interferer

The improvement factor for the TDAF is defined as (7:51)

$$J = \frac{MSE_{TIAF}}{MSE_{TDAF}} \tag{4.1}$$

The improvement factor will be used as the primary figure of merit for the TDAF. A comparison of Bit Error Rate (BER) is also made, but it is emphasized that the results presented are only very rough estimates of the actual BER because only relatively short (8192 symbols) simulations were run. Longer simulation runs were possible, but with a sample frequency of 64 samples per symbol, the run time involved would have reduced the number of runs that could be made. Furthermore, computer disk space was extremely scarce, and the fact that longer runs result in longer data files contributed to the decision to limit the length of each simulation run.

4.2 Simulation in a Noisy Environment

The first runs of the simulations were made with with the carrier amplitude of the interfering signal set to zero. The amplitude of the SOI carrier and the noise gain are shown in Table 4.1. The remaining input parameters were not changed from one simulation to the next. Their values are shown in Tables 4.2 and 4.3.

SNR	Carrier Amplitude	Noise Gain
-20 dB	0.140720	0.995037
-15 dB	0.247602	0.984554
-10 dB	0.426401	0.953463
-5 dB	0.693186	0.871635
0 dB	1.000000	0.707107
5 dB	1.232678	0.490156
10 dB	1.348400	0.301511
15 dB	1.392370	0.175081
20 dB	1.407195	0.095037

Table 4.1. SOI carrier amplitude and noise gain for noisy environment simulations

Input Value	Parameter
6509731	random bit generator seed
-1018	AWGN seed
64	number of samples/symbol
1	Manchester or Bipolar format (1=Bi, 0=Man)
1	Pulse shaping (1=y, 0=n)
64	Num taps in FIRs
32	Num taps in TIAF
32	Num taps in each bank of TDAF
1.0	SOI pulse shaping LPF cutoff freq
1.0	SOI pulse shaping LPF gain
See Table 4.1	SOI carrier amplitude
6.0	SOI carrier frequency
.95	SNOI baud rate
0.0	SNOI carrier amplitude
6.0	SNOI carrier frequency
1.0	output LPF gain
1.0	output LPF cutoff frequency
0.0	demodulator phase shift
See Table 4.1	Noise gain
0.05	misadjustment
1	outputflag (1=random data, 0=square wave)
16	Num symbols in one epoch
512	Num of epochs (does not include adaptation)
32	Number of adaptation epochs

Table 4.2. Input parameters for noisy environment simulations for the BER version

Input Value	Parameter
6509731	random bit generator seed
-1018	AWGN seed
64	number of samples/symbol
1	Manchester or Bipolar format (1=Bi, 0=Man)
1	Pulse shaping (1=y, 0=n)
64	Num taps in FIRs
32	Num taps in TIAF
32	Num taps in each bank of TDAF
1.0	SOI pulse shaping LPF cutoff freq
1.0	SOI pulse shaping LPF gain
See Table 4.1	SOI carrier amplitude
6.0	SOI carrier frequency
.95	SNOI baud rate
0.0	SNOI carrier amplitude
6.0	SNOI carrier frequency
1.0	output LPF gain
1.0	output LPF cutoff frequency
0.0	demodulator phase shift
See Table 4.1	Noise gain
0.05	misadjustment
1	outputflag (1=random data, 0=square wave)
100	Num of epochs
16	Number of symbols to average
512	Number of symbols per epoch

Table 4.3. Input parameters for noisy environment simulations for the LC version

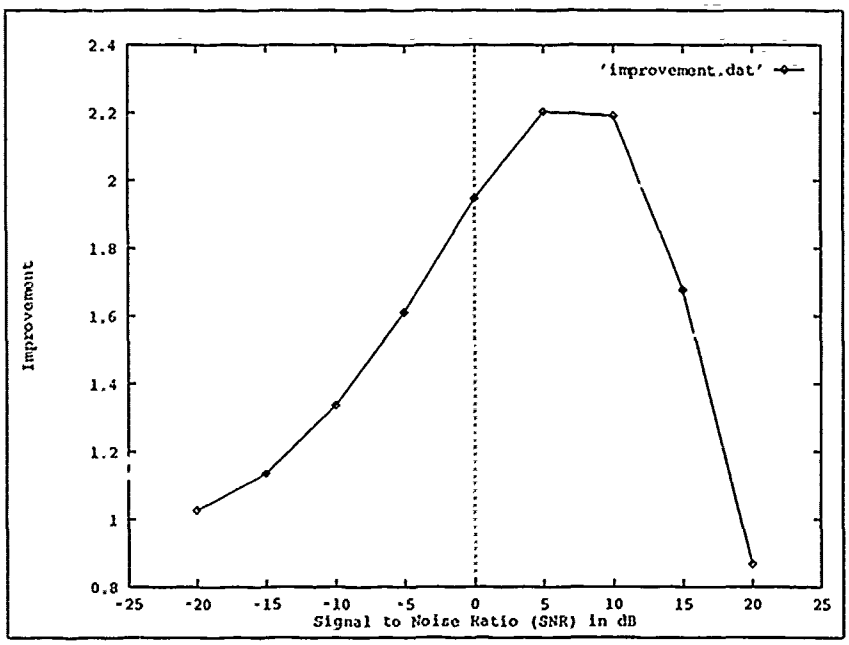


Figure 4.1. Improvement factor as a function of SNR

4.2.1 Improvement Factor. Figure 4.1 shows the improvement possible for varying SNR. The data used was generated by the BER version of the program because that version freezes the filter weights after a specified adaptation time. The data for Figure 4.1 was taken from Table 4.4. Reed shows in his PhD dissertation (7:77) that the TDAF (when fully adapted - what he calls an optimal time dependent adaptive filter) will always do at least as well as the TIAF (also when fully adapted). Why, then, is the MSE smaller for the TIAF than for the TDAF at 20 dB in Table 4.4? The key is that the TDAF could have adapted further if the simulation had not frozen its filter weights. When the same simulation was re-run and the adaptation time was doubled (from 512 adaptation symbols to 1024 adapation symbols), the MSE for the TDAF fell to 0.002970, and the improvement factor rose to 2.11212. Therefore, the shape of the curve in Figure 4.1 is dependent on the amount time the filter is allowed to adapt.

SNR	MSE		J
	TDAF	TIAF	
-20 dB	0.535764	0.550010	1.02659
-15 dB	0.427564	0.484972	1.13427
-10 dB	0.272262	0.363822	1.33629
-5 dB	0.141466	0.227957	1.61139
0 dB	0.064374	0.125292	1.94631
5 dB	0.027251	0.060045	2.20341
10 dB	0.012286	0.026911	2.19038
15 dB	0.007024	0.011781	1.67725
20 dB	0.005233	0.004538	0.86719

Table 4.4. MSE data for simulations in AWGN

The symmetry of Figure 4.1 is not unexpected. When the SNR is high, both filters achieve a nearly optimum solution, resulting in an improvement factor near unity. When the SNR is low, neither filter can "lock on" to the SOI; the MSE for both filters is large, and again, the improvement factor drops to near unity. When the SNR is in the moderate range, however, the TDAF's advantage due to stationarization of the SOI comes into play as illustrated by the larger improvement factor.

4.2.2 Comparison of Filtered Demodulated Signals. A comparison of the demodulated signals for various levels of SNR can be seen in Figures 4.2 through 4.6. Note that the level of the output is reduced each time the SNR is decreased. The reason for that is simple. In order to keep the adaptation time the same for each run, the total amount of power input to the two filters is normalized to 0 dBW (see Table 4.1). Recalling from Section 2.3.2 that the adaptation coefficient, μ, depends on the input power to the filter, maintaining an input power of 0 dBW to the filter results in a constant value for μ for all runs. Hence, the adaptation time is the same.

4.2.3 Comparison of MSE Learning Curves. If the MSE for an adaptive filter is plotted as a function of sample number, the result is the "learning curve" for the filter (10:51). For the TIAF, the data for the learning curve is collected in the

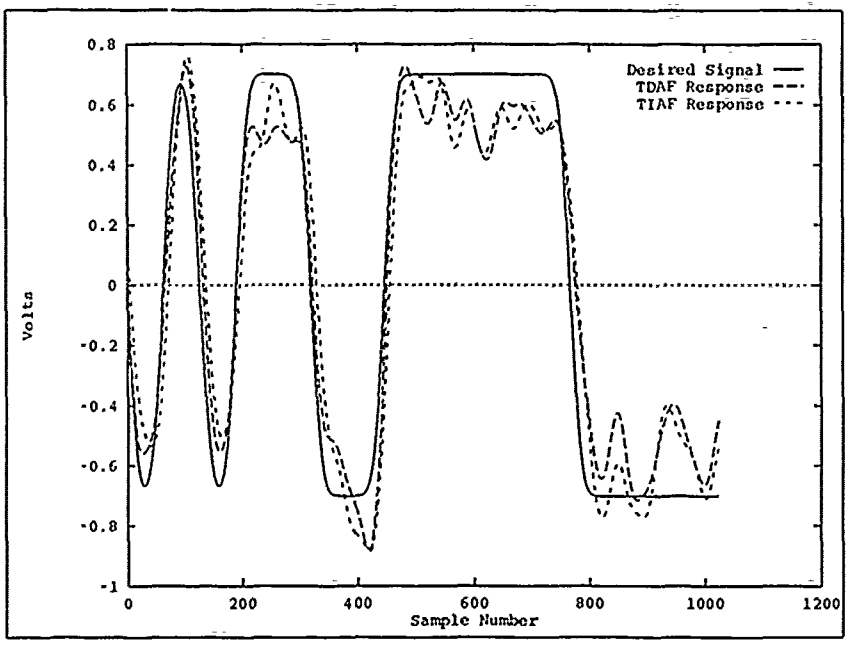


Figure 4.2. The desired signal, TDAF response, and TIAF response for 0 dB SNR

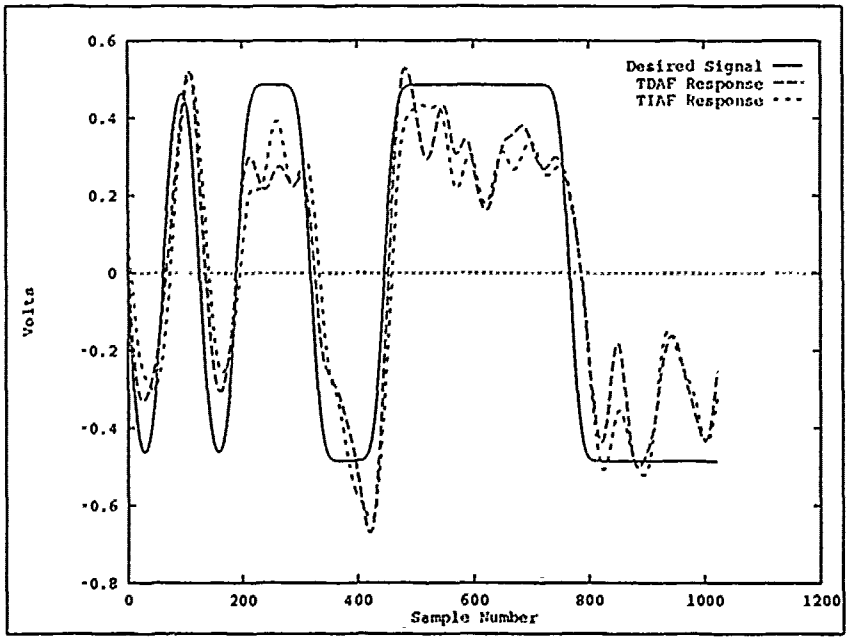


Figure 4.3. The desired signal, TDAF response, and TIAF response for -5 dB SNR

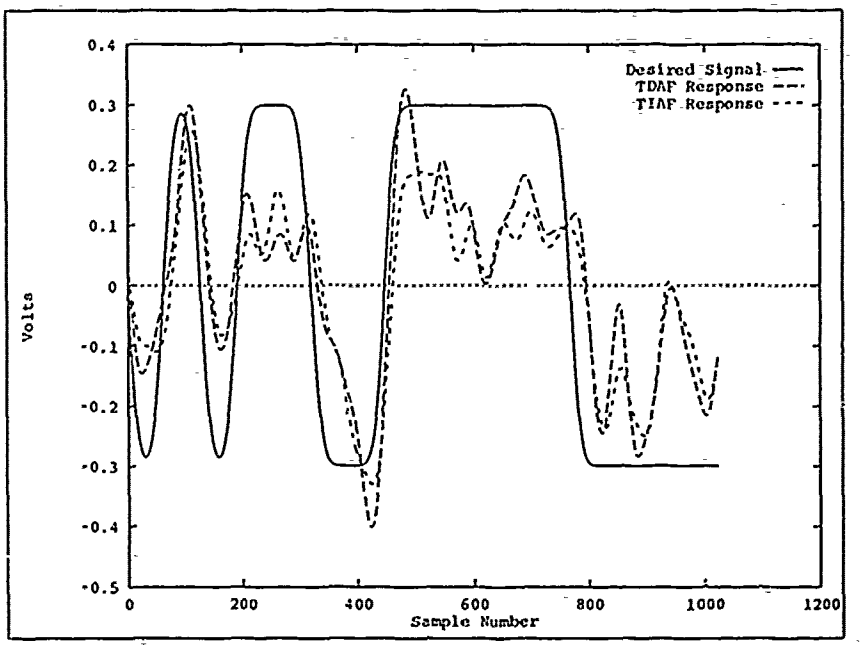


Figure 4.4. The desired signal, TDAF response, and TIAF response for -10 dB SNR

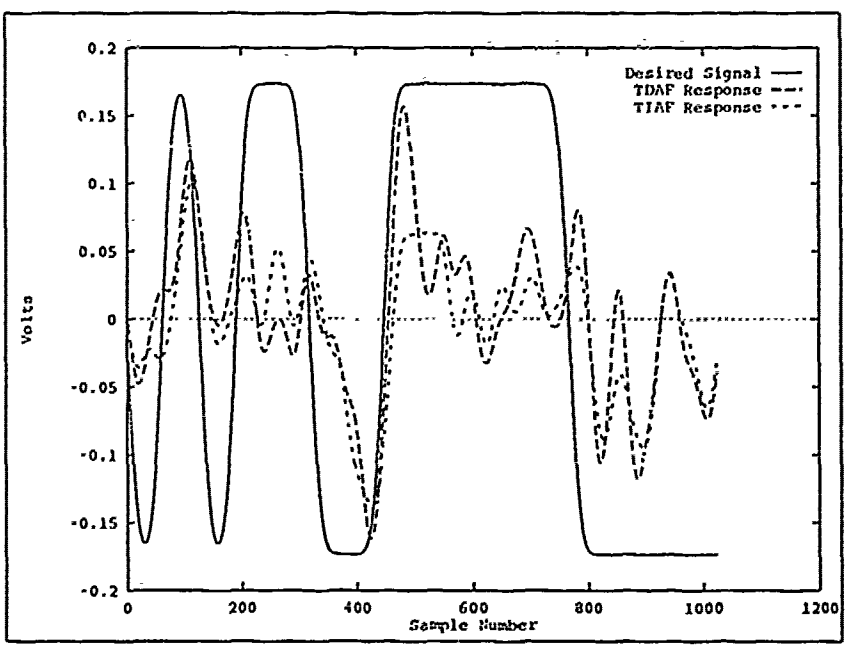


Figure 4.5. The desired signal, TDAF response, and TIAF response for -15 dB SNR

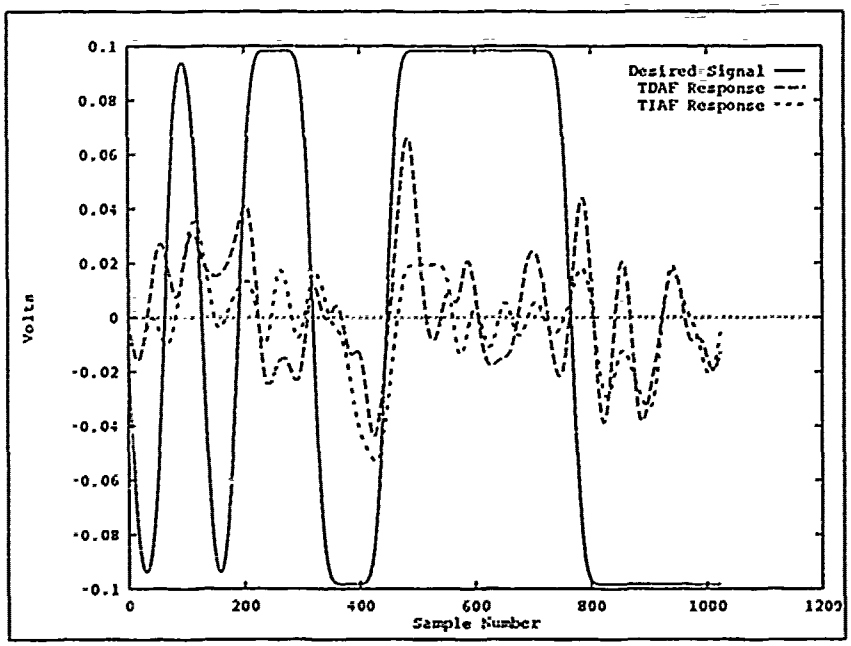


Figure 4.6. The desired signal, TDAF response, and TIAF response for -20 dB SNR

following manner. The filter is initialized and allowed to adapt; the resulting error for each sample is squared and stored in an array. Then the filter is reinitialized, and again allowed to adapt. The resulting error is squared, and added to the previous error data. After this process is repeated M times, where M is some sufficiently large number (M = 100 in this thesis), each element is divided by M. The result is an estimate of the MSE. The larger M, the better the estimate.

For the TDAF, the process is somewhat more complicated. Recalling from Section 2.3.3 and 3.3.7 that the individual TIAFs comprising the TDAF are adapted only once per symbol, it does not make sense to plot the learning curve as a function of absolute sample number. This is because no filter adaptation has occurred from one sample to the next in the TDAF as is does in the TIAF. Instead, the adaptation for the TDAF is from one *symbol* to the next. Therefore, for the TDAF, in addition to averaging the squared error for each reinitialized run to the next as in the TIAF, the error for each sample comprising a given symbol is averaged. The resulting learning

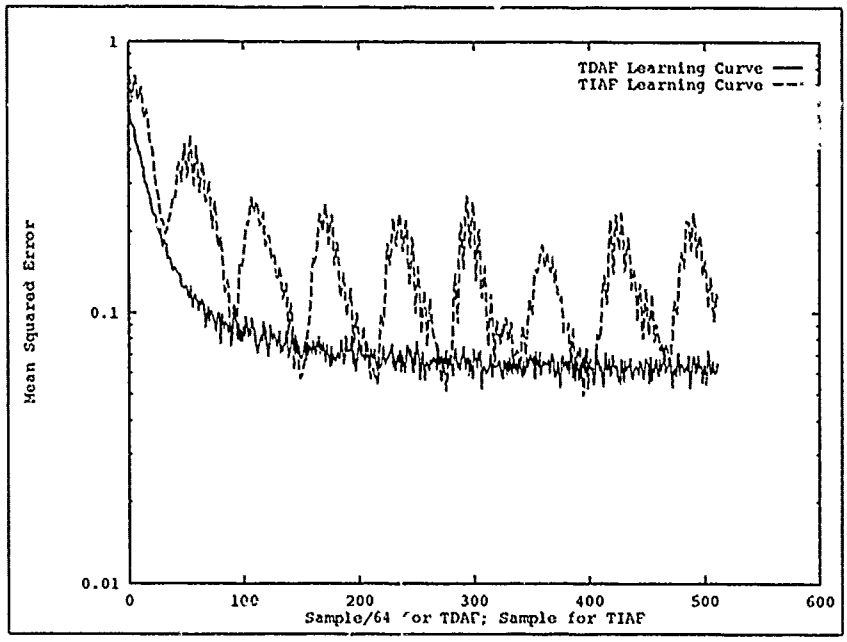


Figure 4.7. MSE learning curves for the TDAF and TIAF for 0 dB SNR.

curve plots the MSE as a function of symbol number, or (sample number)/(samples per symbol).

Referring to Figures 4.7 through 4.11, note that the SNR is proportional to the initial slope of the learning curve. This is because strong correlation between the input signal and the desired signal yields an error surface with steep sides (see Section 2.3.2). Steep error surface sides result in faster adaptation.

The learning curve plots have been normalized by the average power in the desired signal. This was done so that comparisons of the rate of learning could be made without regard to the magnitude of the MSE.

The purpose of the learning curve is to show the adaptation process of the filters. The learning curve plots are not intended to provide an indication of the value of the MSE; instead, refer to Table 4.4.

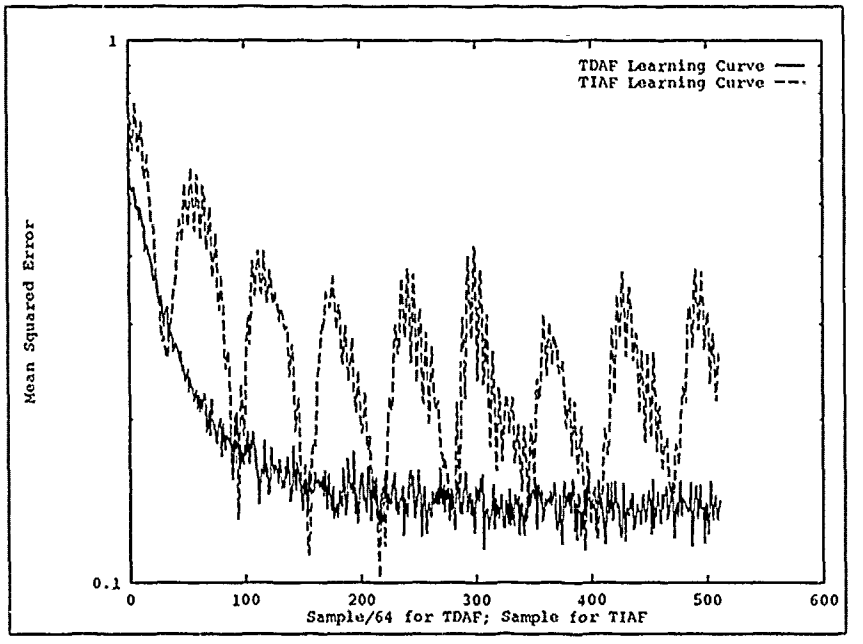


Figure 4.8. MSE learning curves for the TDAF and TIAF for -5 dB SNR

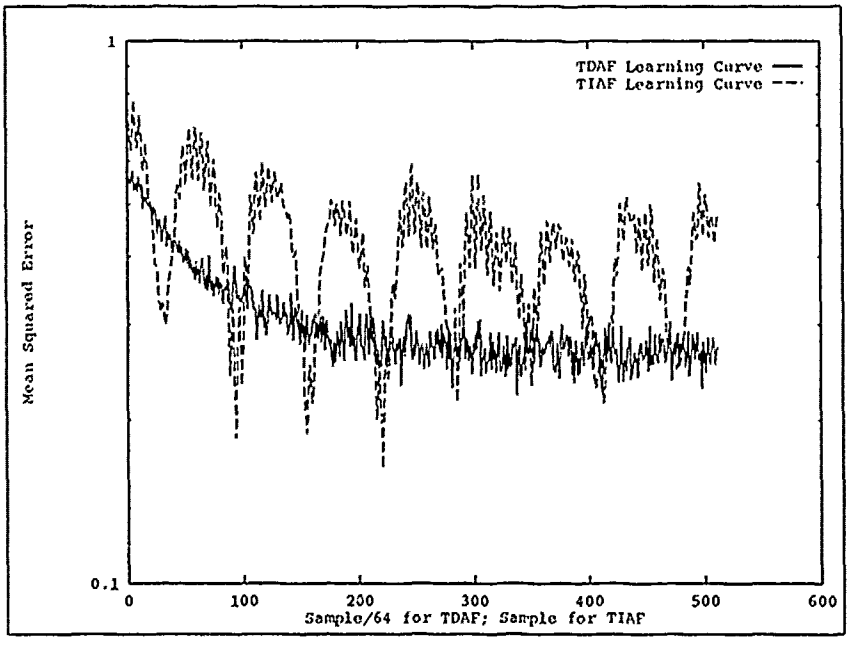


Figure 4.9. MSE learning curves for the TDAF and TIAF for -10 dB SNR

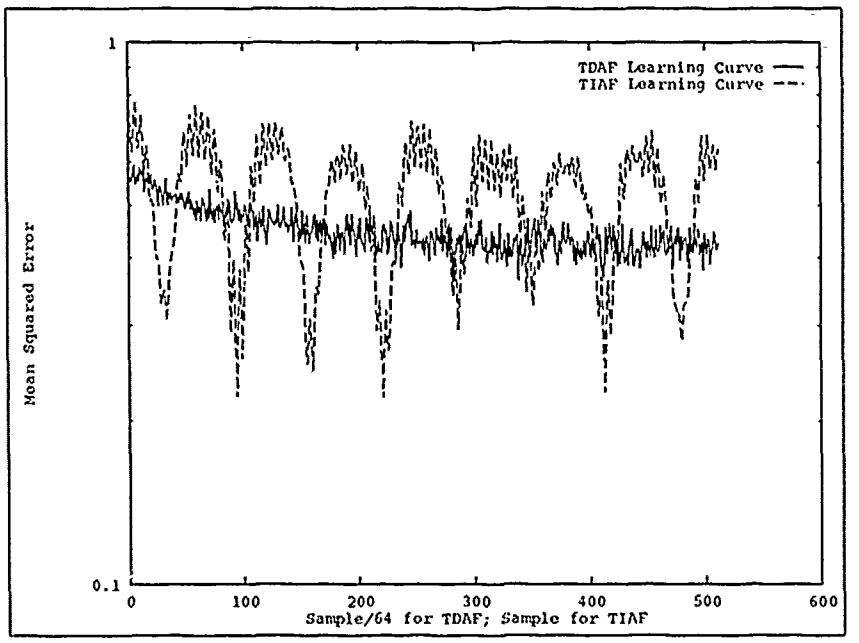


Figure 4.10. MSE learning curves for the TDAF and TIAF for -15 dB SNR

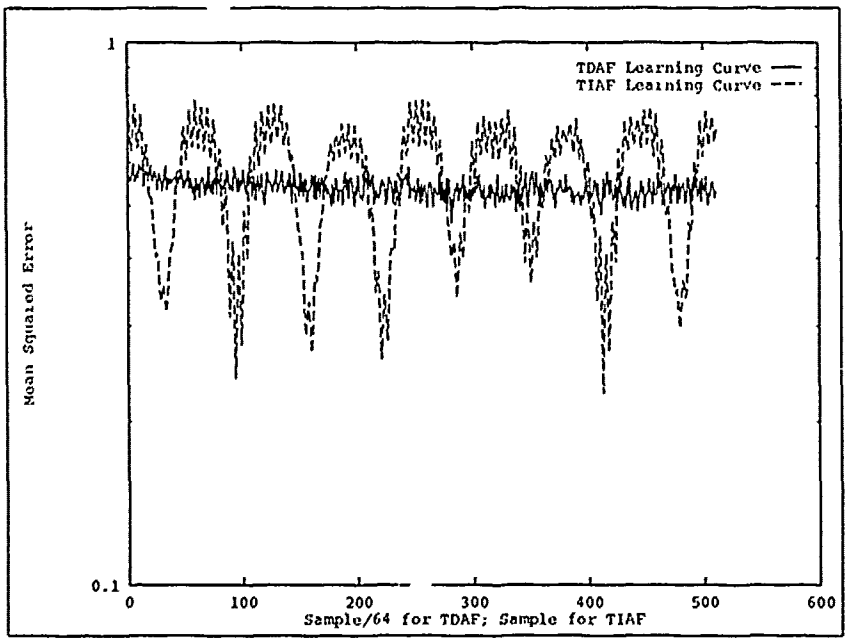


Figure 4.11. MSE learning curves for the TDAF and TIAF for -20 dB SNR

SNR	TDAF	TIAF
0 dB	0.0	0.0
-5 dB	0.004029	0.008058
-10 dB	0.061653	0.070321
-15 dB	0.198877	0.183372
-20 dB	0.535764	0.550010

Table 4.5. BER for five simulations in a noisy environment

4.2.4 Bit Error Rate An error detection occurs when the sum of samples over a symbol time is less than zero when a 1 was transmitted, and greater than zero when a -1 was transmitted. BER is calculated according to

$$\text{BER} = \frac{\text{\# of Bit Errors}}{\text{\# of Symbols in Simulation}} \tag{4.2}$$

Since a BPSK modulator was used, the number of symbols equals the number of bits. In many instances, no errors were detected during the course of the simulation. It is not possible to overemphasize the point that the BER for that particular SNR is not zero. The only conclusion that may be made is that for that particular simulation run, no bit errors were made. With that in mind, refer to Table 4.5. Note that the greatest reduction in BER occurs at -5 dB SNR. These results seem to compare favorably with those plotted in Figure 4.1. Each simulation was 8192 symbols in duration.

4.2.5 Summary The data presented show that the TDAF in general provides some improvement over the TIAF as long as the SNR is no worse than about -10 dB. When the SNR is very high, both filters achieve a low MSE; hence, the improvement is not appreciable.

4.3 Simulation in Interference

In this section the performance of the TDAF is measured under the condition that an interference signal of the same carrier frequency and similar baud rate is

SIR	SOI	SNOI
-20 dB	0.140720	1.407195
-15 dB	0.247602	1.392370
-10 dB	0.426401	1.348400
-5 dB	0.693186	1.232678
0 dB	1.000000	1.000000
5 dB	1.232678	0.693186
10 dB	1.348400	0.426401
15 dB	1.392370	0.247602
20 dB	1.407195	0.140720

Table 4.6. SOI and SNOI carrier amplitude for interference environment simulations

present in the channel. As in Section 4.2, the total power into the filter is maintained at 1 watt. See Table 4.6 for the carrier amplitudes used.

Somewhat unrealisticly, the condition that no AWGN is present on the channel was chosen in an attempt to identify the performance differences of the filters for noise and interference. The ratio of interference baud rate to SOI baud rate was set at 0.95 for all simulations in this section. It was thought that such a ratio would adequately stress the filters without rendering either TDAF or TIAF useless. Surprisingly, the adaptive filters are relatively insensitive to the baud rate of the interferer as can be seen in Figures 4.24 and 4.25. Table 4.7 shows the input parameters used to obtain the data for the simulations using the BER version of the program. Table 4.8 shows the input parameters used for the LC version simulations.

4.3.1 Improvement Factor. The improvement gain for the TDAF was impressive when interference was present. Table 4.9 shows the results obtained from the BER version of the simulation. The data shown has been plotted in Figure 4.12. Note the similarity in shape to Figure 4.1 in Section 4.2.1. In Figure 4.12 however, the magnitude of the improvement is significantly higher than in Figure 4.1. This is an indication that the major advantage of the TDAF is in interference rejection.

Input Value	Parameter
6509731	random bit generator seed
-1018	AWGN seed
64	number of samples/symbol
1	Manchester or Bipolar format (1=Bi, 0=Man)
1	Pulse shaping (1=y, 0=n)
64	Num taps in FIRs
32	Num taps in TIAF
32	Num taps in each bank of TDAF
1.0	SOI pulse shaping LPF cutoff freq
1.0	SOI pulse shaping LPF gain
See Table 4.6	SOI carrier amplitude
6.0	SOI carrier frequency
.95	SNOI baud rate
See Table 4.6	SNOI carrier amplitude
6.0	SNOI carrier frequency
1.0	output LPF gain
1.0	output LPF cutoff frequency
0.0	demodulator phase shift
0.0	Noise gain
0.05	misadjustment
1	outputflag (1=random data, 0=square wave)
16	Num symbols in one epoch
512	Num of epochs (does not include adaptation)
32	Number of adaptation epochs

Table 4.7. Input parameters for interference environment simulations for the BER version

Input Value	Parameter
6509731	random bit generator seed
-1018	AWGN seed
64	number of samples/symbol
1	Manchester or Bipolar format (1=Bi, 0=Man)
1	Pulse shaping (1=y, 0=n)
64	Num taps in FIRs
32	Num taps in TIAF
32	Num taps in each bank of TDAF
1.0	SOI pulse shaping LPF cutoff freq
1.0	SOI pulse shaping LPF gain
See Table 4.6	SOI carrier amplitude
6.0	SOI carrier frequency
.95	SNOI baud rate
See Table 4.6	SNOI carrier amplitude
6.0	SNOI carrier frequency
1.0	output LPF gain
1.0	output LPF cutoff frequency
0.0	demodulator phase shift
0.0	Noise gain
0.05	misadjustment
1	outputflag (1=random data, 0=square wave)
100	Num of epochs
16	Number of symbols to average
512	Number of symbols per epoch

Table 4.8. Input parameters for interference environment simulations for the LC version

SNR	MSE		J
	TDAF	TIAF	
-20 dB	0.455682	0.954006	2.09358
-15 dB	0.269755	0.870597	3.22736
-10 dB	0.096536	0.741551	7.68160
-5 dB	0.039515	0.570294	14.43234
0 dB	0.019807	0.370759	18.71878
5 dB	0.011584	0.183849	15.87094
10 dB	0.008921	0.072100	8.08285
15 dB	0.009487	0.028873	3.04343
20 dB	0.007489	0.012307	1.64334

Table 4.9. MSE data for simulations in interference

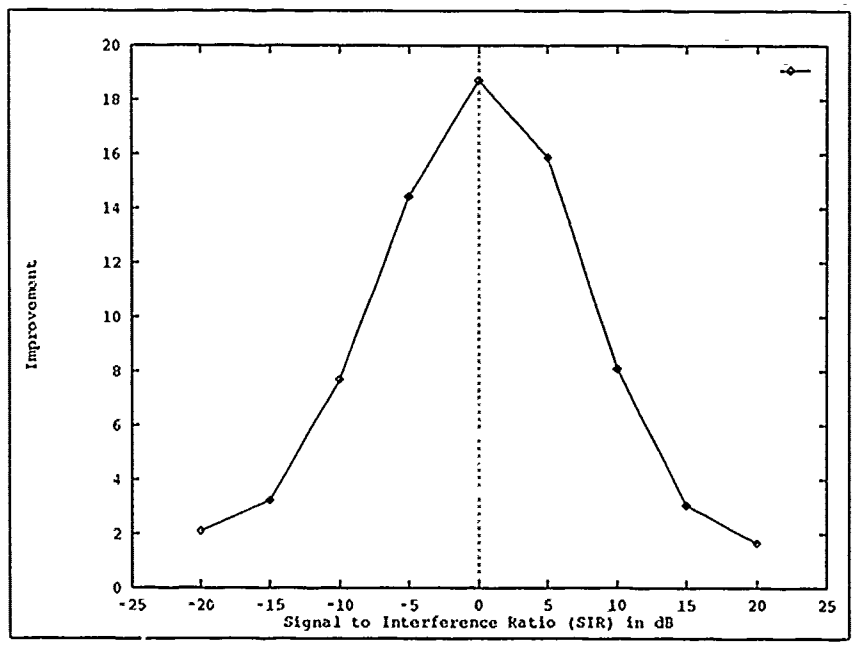


Figure 4.12. Improvement factor as a function of SIR

4.3.2 Comparison of Filtered Demodulated Signals. A comparison of the demodulated signals for various levels of SIR can be seen in Figures 4.13 through 4.17. Note that as the SIR gets smaller, the response of the TIAF loses all similarity to the desired signal. For the TDAF, on the other hand, the shape of the response is very nearly correct even at -20 dB SIR, although the level of its output is not. Figure 4.18 is a plot of the SNOI over the same time interval as Figures 4.13 through 4.17. Note the similarity of the TIAF output in Figures 4.16 and 4.17 to the plot in Figure 4.18 indicating that the TIAF could not track the SOI, and instead passed the SNOI.

4.3.3 Comparison of MSE Learning Curves. The learning curves shown in Figures 4.19 through 4.23 are computed in the same fashion as those in Section 4.2.3. Again, the adaptive filters converge more rapidly when the SIR is high. In fact Figures 4.22 and 4.23 show that there is little reduction in the level of the MSE throughout the course of the simulation.

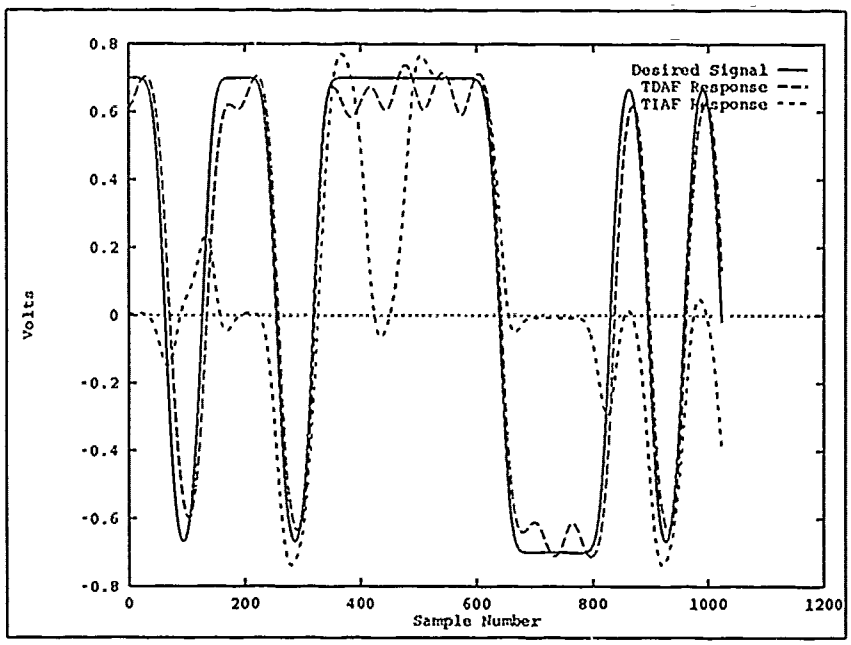


Figure 4.13. The desired signal, TDAF response, and TIAF response for 0 dB SIR

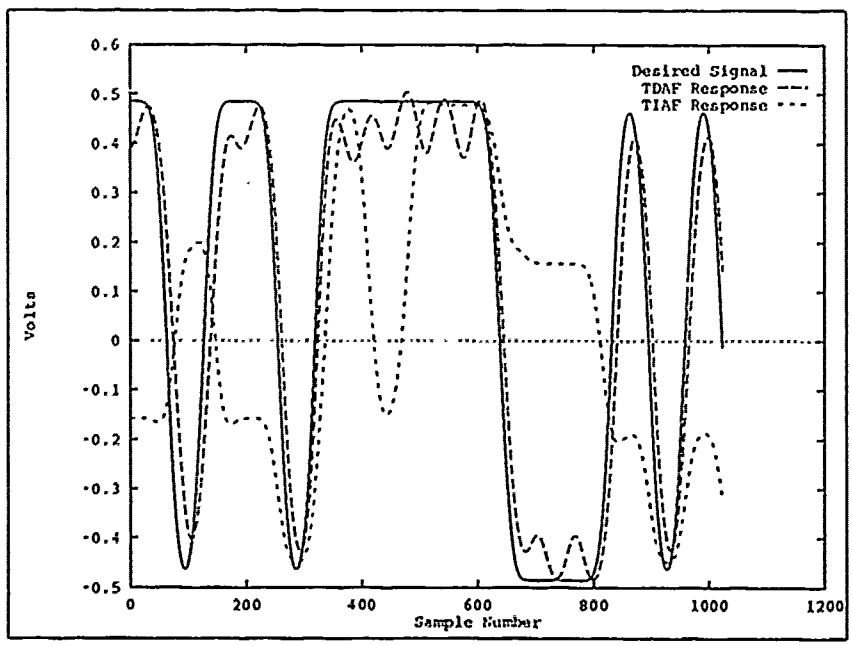


Figure 4.14. The desired signal, TDAF response, and TIAF response for -5 dB SIR

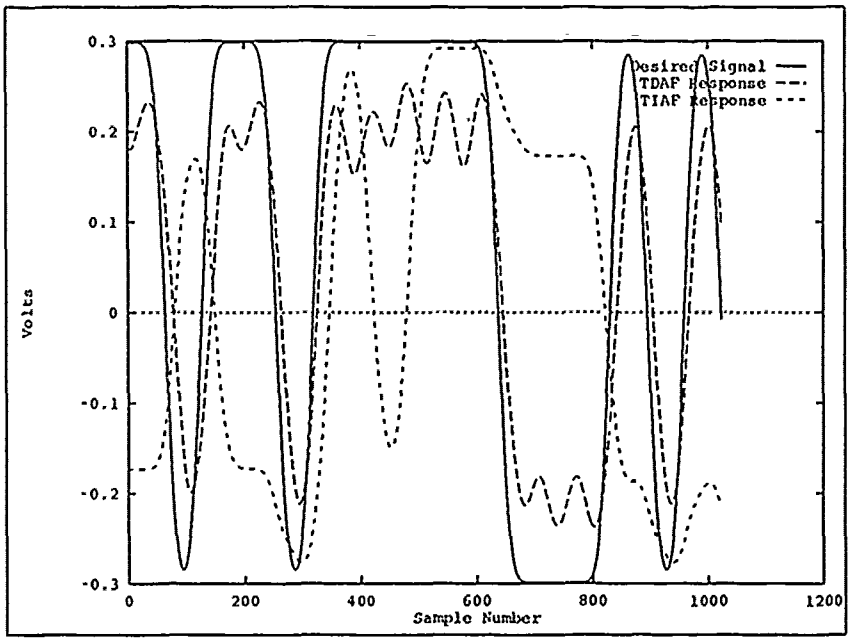


Figure 4.15. The desired signal, TDAF response, and TIAF response for -10 dB SIR

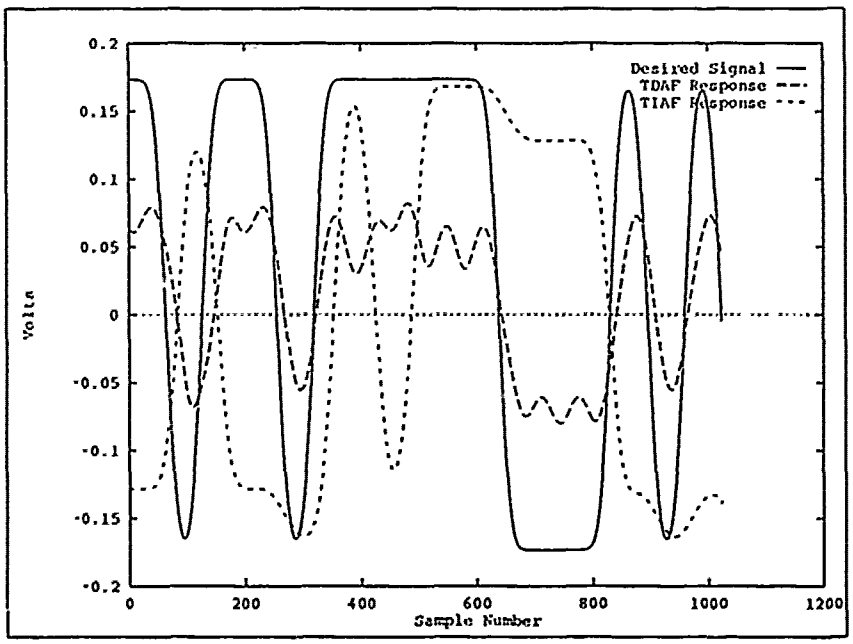


Figure 4.16. The desired signal, TDAF response, and TIAF response for -15 dB SIR

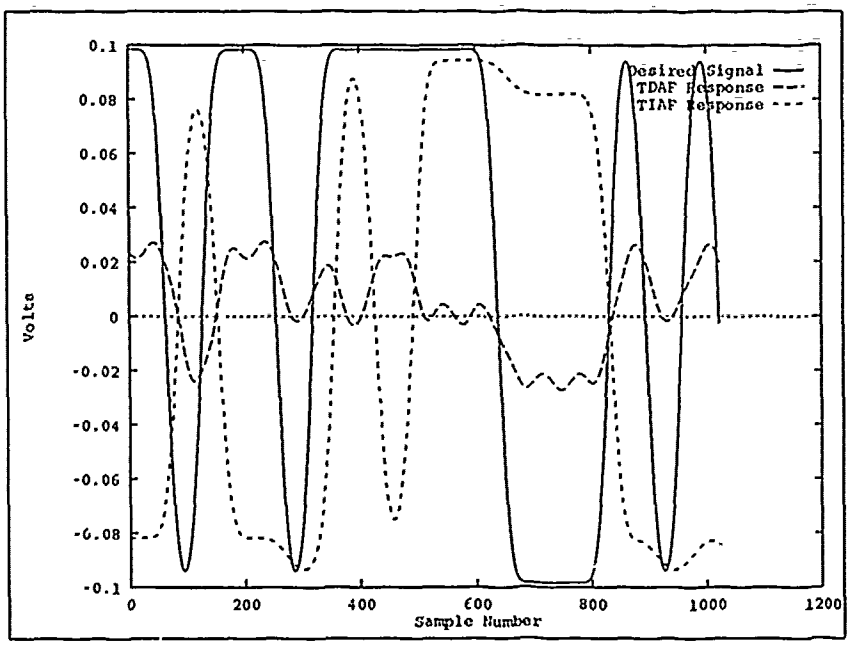


Figure 4.17. The desired signal, TDAF response, and TIAF response for -20 dB SIR

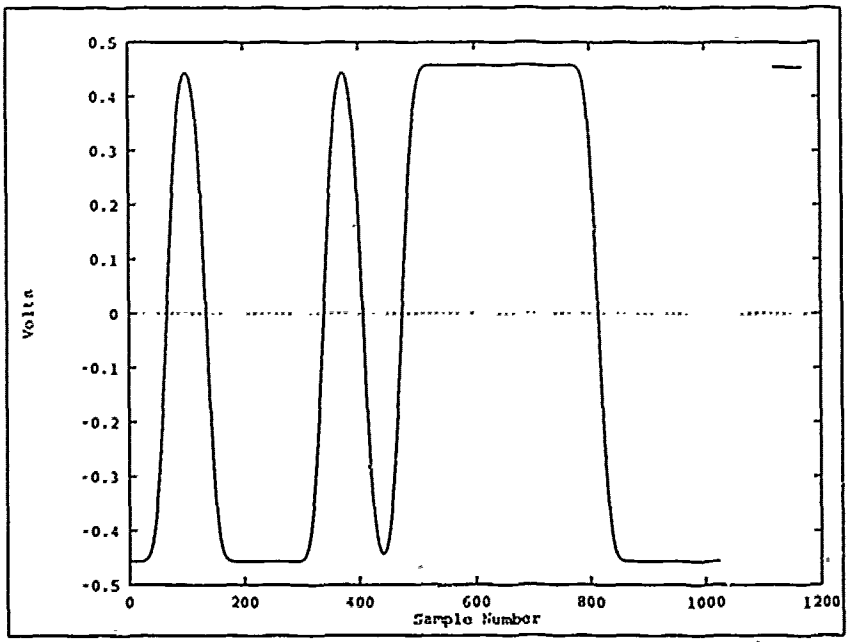


Figure 4.18. The SNOI present in Figures 4.13 through 4.17

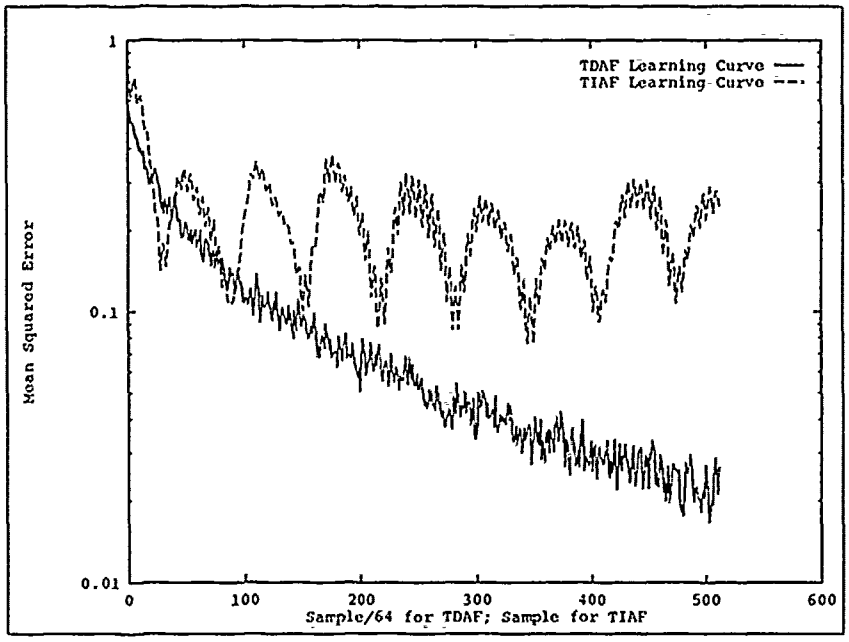


Figure 4.19. MSE learning curves for the TDAF and TIAF for 0 dB SIR

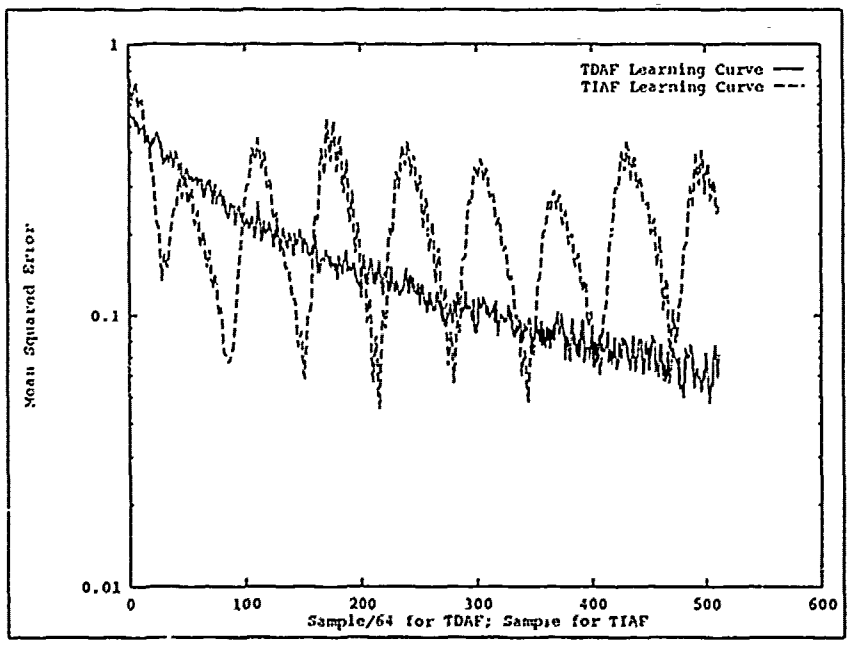


Figure 4.20. MSE learning curves for the TDAF and TIAF for -5 dB SIR

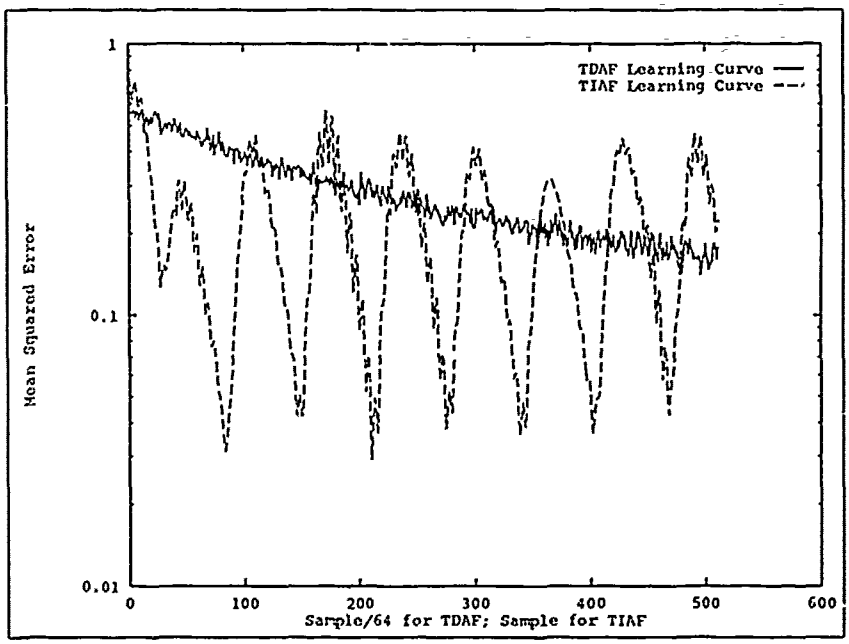


Figure 4.21. MSE learning curves for the TDAF and TIAF for -10 dB SIR

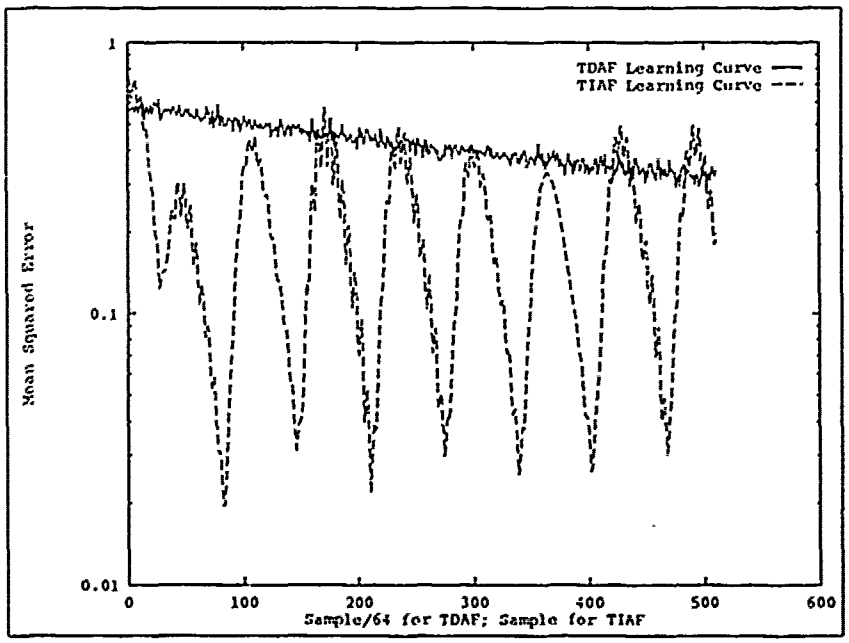


Figure 4.22. MSE learning curves for the TDAF and TIAF for -15 dB SIR

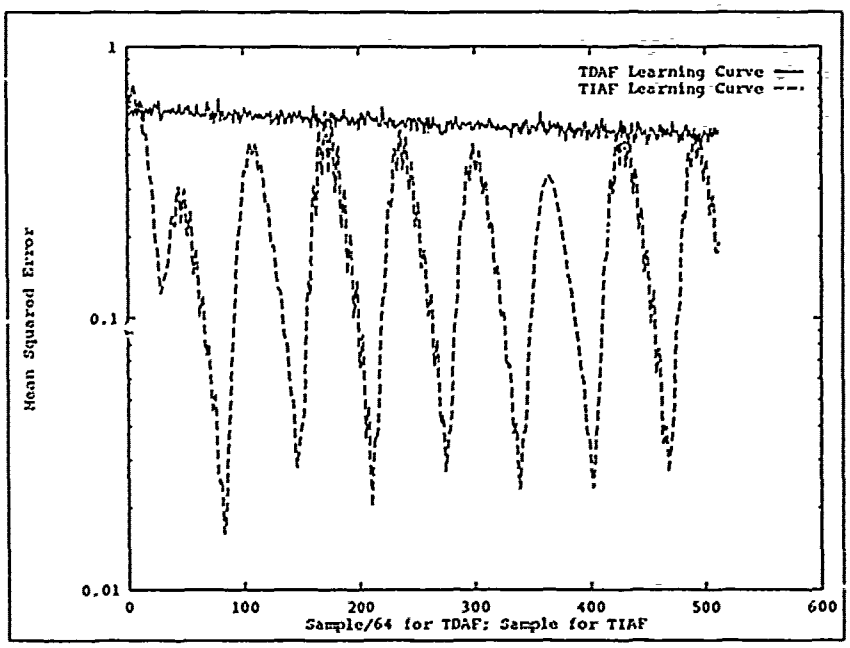


Figure 4.23. MSE learning curves for the TDAF and TIAF for -20 dB SIR

SIR	TDAF	TIAF
0 dB	0.0	0.220974
-5 dB	0.0	0.396044
-10 dB	0.0	0.449152
-15 dB	0.0	0.467708
-20 dB	0.152240	0.481016

Table 4.10. BER for five Simulations in Interference

4.3.4 Bit Error Rate Table 4.10 shows the BER observed during the simulations in interference. These five simulations indicate that there may be a significant improvement in BER when a TDAF is used to extract a signal from a strong interferer . Referring to Table 4.5, note that in noise, both filters suffered approximately 50% error rate. In interference, however, the BER for the TDAF dropped from approximately 50% to under 20%.

4.3.5 Summary. The improvement obtained for simulations in an interference were better overall than for those run in a noisy environment. The improvement

factor was higher at all levels of SIR, and the Bit Error Rate appeared to be lower. The simulations show that the TDAF is very good at rejecting interference, even when the interferer is at the same carrier frequency.

4.4 Varying the Baud Rate of the Interferer

The data used to produce Figure 4.24 and 4.25 was compiled from 12 simulations. A simulation was run for each the four relative baud rates at -10 dB, 0 dB and 10 dB SIR. The carrier amplitudes for those levels of SIR can be found in Table 4.6. All remaining input parameters were the same as those listed in Table 4.7.

The data in this section support the surprising result that the TDAF is essentially unaffected by the baud rate of the interferer. Figure 4.24 illustrates the TDAF's ability to reject an SNOI even when its baud rate, modulation type and carrier frequency are the same as the SOI. Figure 4.25 is included to show the TIAF's performance under the same conditions. The TIAF was also fairly insensitive to the baud of the interferer, but note that the MSE for the TIAF is approximately 10 dB worse.

4.5 Varying the Carrier Frequency of the SNOI

The data for Figure 4.26 were compiled from five simulations. In each simulation, the carrier frequency of the SNOI was varied as indicated. The SIR was fixed at 0 dB. The remaining input parameters to the simulation were set according to Table 4.7.

The TDAF is not affected by SNOI carrier frequency proximity as illustrated by Figure 4.26. The TIAF however suffers a significant increase in MSE as the carrier frequencies of the SOI and SNOI coincide.

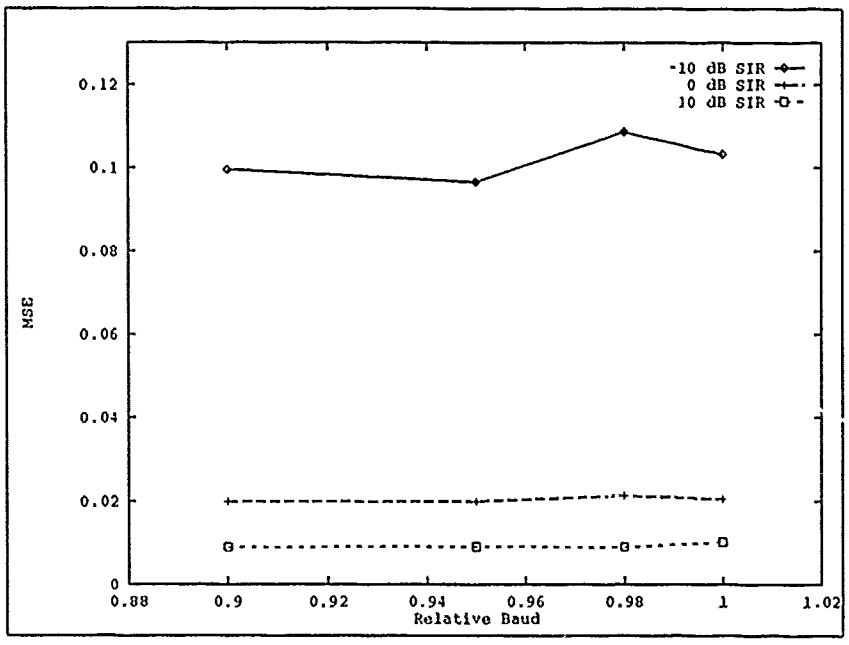


Figure 4.24. MSE as a function relative baud rate at various SIRs for the TDAF

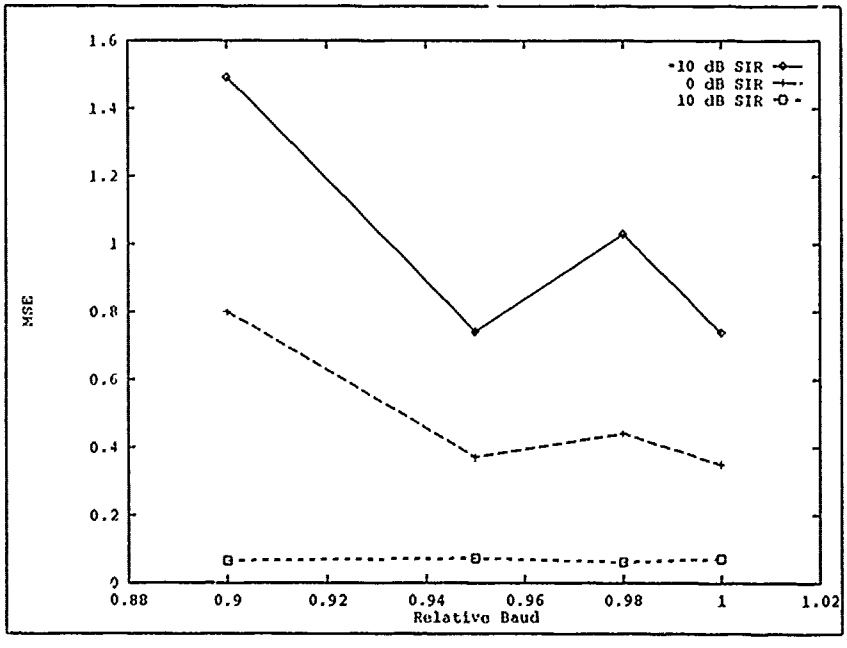


Figure 4.25. MSE as a function relative baud rate at various SIRs for the TIAF

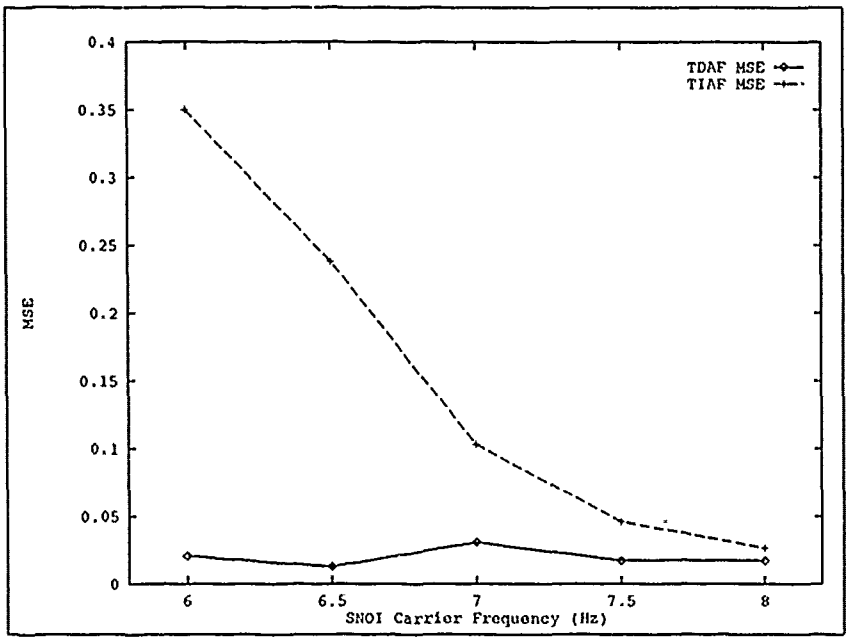


Figure 4.26. MSE as a function SNOI carrier frequency. SOI carrier frequency fixed at 6 Hz

4.6 Chapter Summary

Over 50 simulations were run to obtain the data for this chapter. The results of those simulations indicate that that for a given communications system, if a choice exists between a TDAF and a TIAF, then the conditions under which the filter will be operating must be considered prior to making the selection. The simulations yielded improvement factors that ranged from -0.6 dB to 12 dB. Even though there was one case where the TIAF outperformed the TDAF, it was shown that given sufficient adaptation time, the TDAF will always do at least as well as the TIAF.

V. Conclusions and Recommendations

5.1 Conclusions

This thesis presented a TDAF which could be used to improve the signal to interference ratio (SIR) and signal to noise ratio (SNR) of digitally modulated communications signals. The performance improvement of the TDAF over the TIAF was determined based on the application of various metrics, including Mean Square Error (MSE) and Bit Error Rate (BER). The collected data indicate that there are advantages and disadvantages of the TDAF. The advantages can be summarized as follows:

- The improvement in MSE obtained by using a TDAF over a TIAF in a noisy environment can be significant. With a 0 dB SNR, a performance gain of nearly 3 dB can be expected.
- The improvement in MSE obtained by using a TDAF over a TIAF in situation where there is a strong interference signal can be even more significant. With a 0 dB SIR, a performance gain of more than 12 dB can be expected.
- A receiver fitted with a TDAF which has been given adequate time to adapt is likely to have a lower BER than a receiver using a TIAF.
- The TDAF always does at least as well as the TIAF, given adequate time to adapt.

The disadvantages are:

- The TDAF takes considerably longer to adapt than the TIAF. While the TIAF adapts at each sample time, the TDAF actually only "fractionally" adapts at each sample time. A full symbol is required to adapt each TIAF in the TDAF structure.

- The TDAF would require much more "real estate" if implemented in silicon. This is due to the multiplicity of filter coefficients required for the TDAF. Fortunately, there is little added computational overhead, since the process of updating filter coefficients is no more frequent for the TDAF than it is for the TIAF.

- The TDAF is somewhat more complex than the TIAF. This should be clear since the TDAF is made up of a bank of parallel TIAFs.

- As SNR at the filter input decreases, the improvement in Mean Square Error decreases. By the time the SNR has degraded to less than -10 dB, the performance gain has dropped to under 1 dB. If that is the case, then given the other disadvantages of the TDAF, the use of a TIAF might make more sense.

5.2 Recommendations

There are several topics that could (and probably should) have been covered in the execution of this research. Some of the more urgent are listed below.

1. Investigate the role that the modulation type plays in the performance gain. By implementing different modulator/demodulator pairs, the simulation can be used to determine improvement gain as a function of modulation type. Other modulation schemes might include:

    - Quadrature Phase Shift Keying (QPSK)
    - Offset Quadrature Phase Shift Keying (OQPSK)
    - M-ary Frequency Shift Keying (FSK)
    - Minimum Shift Keying (MSK)
    - Quadrature Amplitude Modulation (QAM)

2. Implement the TDAF with no external training signal. With some added complexity, the inconvenience of having to transmit a known preample can be circumvented (7), and the filter can be allowed to adapt continuously.

3. Implement the TDAF in the frequency domain. The advantage of doing so can be significant. The cycle frequencies of the signal used in this simulation were at $\pm 2f_c$. In the frequency domain, the optimum TDAF is implemented with as many TIAFs as exist cycle frequencies (7:103). One advantage of fewer TIAFs may be more rapid adaptation since there would be fewer weight vectors to update.

4. Precalculate an initial weight vector for the adaptive filters. Both TDAF and TIAF are initialized with zero weight vectors in the simulation presented here. It should be possible to calculate a "near optimum" solution for the adaptive filters for a given SOI, and then allow the filter weights adapt from that point.

5. Investigate the apparent inconsistency that exists between the MSE returned for the LC version and the MSE returned for the BER version. While the BER consistantly returns smaller Bit Error Rate numbers and MSE for the TDAF, the MSE returned by the LC version is considerably higher for the TDAF.

6. Determine the improvement of the TDAF over the TIAF for the important case of multiple interferers.

7. Determine the improvement of the TDAF over the TIAF under the conditions of simultaneous interference and noise.

Appendix A. *Input Parameters and Output Files for the BER Version*

The BER version of the simulation prompts the user for the following input parameters at run time:

- *Random bit generator seed.* Seeds the random bit generator.
- *Random number generator seed.* Seeds the AWGN generator.
- *Data Sample Rate.* The number of samples per bit of the data.
- *Format.* A flag to select between bi-polar and bi-phase baseband format.
- *Pulse shaping.* A flag to allow or disallow bandlimiting of the baseband data signal.
- *FIR taps* The number of coefficients in the non-adaptive FIR filters. All FIR filters in the simulation then have the same number of taps.
- *TIAF taps.* The number of filter coefficients for the time independent adaptive filter. Stated another way, the number of elements of **W**.
- *TDAF taps.* The number of filter coefficients for the time dependent adaptive filter. See Section 2.3.3
- *Bandlimiting LPF cutoff frequency.* Passed to the simulation in hertz. Both SOI and SNOI channels use the same cutoff.
- *Bandlimiting LPF gain.* Allows adjustment of the level of the output of the pulse shapers.
- *SOI carrier amplitude.*
- *SOI carrier frequency.*

- *SNOI symbol frequency.* Expressed as a fraction of the SOI symbol frequency of 1 symbol per second. May be any positive real number.
- *SNOI carrier amplitude.*
- *SNOI carrier frequency.*
- *Output LPF gain.* Allows adjustment of the demodulator output level.
- *Output LPF cutoff frequency.* In hertz.
- *Demodulator phase shift.* Allows for adjustment of the phase of the bandpass signal into the demodulator.
- *Noise gain.* Allows for adjustment of noise power in the unfiltered bandpass signal.
- *Misadjustment.* See Section 2.3.3
- *Data type fl. q.* Allows for selection between a square wave baseband signal or random data.
- *Symbols per epoch.* The program writes this number of symbols to a data file at the end of a simulation run.
- *Number of run epochs.* Used to specify the number of symbols for the entire simulation (total symbols = symbols per epoch × number of run epochs).
- *Number of adaptation epochs.* Used to specify the length of time the filters are allowed to adapt.

Each run of the BER Version produces six data files. The files are written into the current directory. They are:

- *tdafvolt.dat* The output of the TDAF over the last specified number of symbols (see above).

- *tiafvolt.dat* The output of the TIAF over the same period.
- *dsrdvolt.dat* The desired filter output over the same period.
- *tdafvec.dat* The error signal of the TDAF over the same period.
- *tiafvic.dat* The error signal of the TIAF over the same period.
- *numbers.tex* A file that contains various data and figures of merit for the simulation run:

    1. The adaptation coefficient for the simulation (see Section 2.3.3).
    2. The number of adaptation epochs in the simulation.
    3. The total number of epochs in the simulation.
    4. The number of post-adaptation symbols in the simulation.
    5. The mean squared error for the TDAF.
    6. The mean squared error for the TIAF.
    7. The number of symbol errors for the TDAF.
    8. The number of symbol errors for the TIAF.

Appendix B. *Input Parameters and Output Files for the LC Version*

The LC version of the simulation prompts the user for the following input parameters at run time:

- *Random bit generator seed.* Seeds the random bit generator.
- *Random number generator seed.* Seeds the AWGN generator.
- *Data Sample Rate.* The number of samples per bit of the data.
- *Format.* A flag to select between bi-polar and bi-phase baseband format.
- *Pulse shaping.* A flag to allow or disallow bandlimiting of the baseband data signal.
- *FIR taps* The number of coefficients in the non-adaptive FIR filters. All FIR filters in the simulation then have the same number of taps.
- *TIAF taps.* The number of filter coefficients for the time independent adaptive filter. Stated another way, the number of elements of W.
- *TDAF taps.* The number of filter coefficients for the time dependent adaptive filter. See Section 2.3.3
- *Bandlimiting LPF cutoff frequency.* Passed to the simulation in hertz. Both SOI and SNOI channels use the same cutoff.
- *Bandlimiting LPF gain.* Allows adjustment of the level of the output of the pulse shapers.
- *SOI carrier amplitude.*
- *SOI carrier frequency.*

- *SNOI symbol frequency.* Expressed as a fraction of the SOI symbol frequency of 1 symbol per second. May be any positive real number.
- *SNOI carrier amplitude.*
- *SNOI carrier frequency.*
- *Output LPF gain.* Allows adjustment of the demodulator output level.
- *Output LPF cutoff frequency.* In hertz.
- *Demodulator phase shift.* Allows for adjustment of the phase of the bandpass signal into the demodulator.
- *Noise gain.* Allows for adjustment of noise power in the unfiltered bandpass signal.
- *Misadjustment.* See Section 2.3.3
- *Data type flag.* Allows for selection between a square wave baseband signal or random data.
- *Number of epochs.* The greater the number of epochs, the better the estimate of the MSE.
- *Number of symbols to average.* The MSE from this number of symbols will be averaged at the end of each epoch to provide an estimate of the MSE.
- *Number symbols per epoch.* This number of symbols will be applied to the TDAF and TIAF each epoch.

Each run of version LC produces three data files. The files are written into the current directory. They are:

- *tdaflrn.dat* An estimate of the expected value of the MSE as a function of time for the TDAF.

- *tiaflrn.dat* An estimate of the expected value of the MSE as a function of time for the TIAF.

- *lnumbers.tex* A file that contains various data and figures of merit for the simulation run:

  1. The adaptation coefficient.
  2. The number of symbols represented by the adaptation period.
  3. The mean squared error of the TDAF for the last n symbols of the epoch (where n is selectable at run time).
  4. The mean squared error of the TIAF for the last n symbols of the epoch (where n is selectable at run time).

Appendix C. *Source Code for the BER Version*

```
/*************************************************************************/
/****** This program is a simulated digital communication system ******/
/*************************************************************************/
/*************************************************************************/
/*
/   5 Oct 91:  This version of the program has an optimized TDAF.
/   The filter now has a one dimensional vector (rather than  a matrix
/   as in earlier versions) to hold the values of the input.  I made
/   the startling (to me) observation that each row of the input
/   matrix was identical.
/   7 Oct 91:  Added input parameter "frozen" to the adaptive
/   filters.  When frozen = 0, weights are allowed to adapt.
/   After the filters have been given the specified length of time
/   to adapt, then frozen is set to 1, and filter weights are no
/   longer allowed to adapt.
/   Executable:  frzloop
/                                                                    */
/*************************************************************************/

#define sparc 1       /*   */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>

#define pi                (4*atan(1.0))
#define data_freq         1.0
#define data_rate         1.0/datafreq
#define sqr(x)            (x)*(x)


/*************************************************************************/
/********************** Function Prototypes ******************************/
/*************************************************************************/

double setcutoff(double samplerate,
                 double cutoff);

void  calcfilterweights(int numdelays,
                        double omega_c,
```

```
                        dcuble *contents,
                        double *weights);

int irbit2(unsigned long *bseed);

int datagen(double         time,
            double         datarate,
            int            lastdata,
            double         *lasttime,
            unsigned long  *bseed,
            int            outputflag);

double lpf(double      input,
           int         numdelays,
           double      *weights,
           double      *contents,
           double      gain);

double modulate(double input,
                double carrierfreq,
                double  carrierampl,
                double time);

void inittiaf(double *weights,
              double *contents,
              int  numtaps,
              double *error,
              double *bias);

double tiaf(float input,
            float desired,
            float mu,
            int  numtaps,
            double *error,
            double *weights,
            double *contents, double *bias,
            int frozen);

double demodulate(double input,
                  double carrierfreq,
                  double  phase,
                  double time);

float ran1(int *idum);
```

```
float gasdev(int *idum);

double **inittdafwts(int sampersym,
                     int numtaps);

double *inittdafconts(int numtaps);

double tdaf(double input,
            double desired,
            double mu,
            int sampersym,
            int numtaps,
            double **tdafwts,
            double *tdafconts,
            double *error,
            double *bias,
            int frozen);

double *initerror(int sampersys);

double *initbias(int sampersym);

void nrerror(char error_text[]);

double *dvector(int nl,
                int nh);

double **dmatrix(int nrl,
                 int nrh,
                 int ncl,
                 int nch);

float *vector(int nl,
              int nh);

int manchester(int input,
               double time,
               double datarate,
               double  lasttime);

void free_dvector(double *v,
                  int nl,
                  int nh);

void free_dmatrix(double **m,
```

```
                int nrl,
                int nrh,
                int ncl,
                int  nch);

int bipolar(int input);

/*****************************************************************************/


int main() {


   unsigned long
       numsamples,
       bseed,
       bitvar=0;

   int numsymbols,
       numfirtaps,
       numtiaftaps,
       numtdaftaps,
       numfirdelays,
       numtiafdelays,
       numloops,
       samplefreq,
       count,
       outputflag,
       adaploops,
       sample,
       sample1,
       loopnum,
       d1t=1,
       d2t,
       d3t=1,
       d4t,
       d5t,
       d6t,
       sampersym,
       format,
       shape,
       tdafbiterror=0,
       tiafbiterror=0,
       offsetfound=0,
       xsample,
```

```
    bitcount = 0,
    avgloops,
    msecount = 0,
    nseed,
    frozen=0;

double
   samplerate,
   datafreq    = data_freq,
   datarate    = data_rate,
   carrierfreq,
   carrierampl,
   noisegain,
   misadjust,
   mu,
   deltaphase,
   gain1,
   gain2,
   tiafgain2,
   avg = 0.0,
   tiafavg = 0.0,
   lpfin_cutoff,
   lpfout_cutoff,
   ilpfin_cutoff,
   *lpfconts,
   *lpfwts,
   *tiafwts,
   *tiafconts,
   *nnwts,
   *nnconts,
   *outweights,
   *outcontents,
   *tiafoutweights,
   *tiafoutcontents,
   tiaferror = 0.0,
   tiafbias = 0.0,
   omega_c,
   omega_c1,
   time = 0.0,
   s1t, s2t, s3t,
   s4t, s5t, s6t,
   s7t, s8t,
   x6t, x7t, x8t,
   nn1t, nn2t,
   tdafbitnum=0.0,
```

```
   tiafbitnum=0.0,
   dsrdbitnum=0.0,
   s3tlast,
   nn2tlast,
   lasttime,
   **tdafwts,
   *tdafconts,
   *tdaferror,
   *tdafbias,
   idatarate,
   idatafreq,
   ilasttime,
   *ilpfconts, *ilpfwts,
   igain1,
   icarrierfreq,
   icarrierampl,
   itime = 0.25,
   iomega_c,
   desiredpower,
   interfpower;
float
   *mse, *tempmse,
   *tiafmse, *tiaftempmse,
   adapfactor=0.1178,
   loopfactor;
char
   buffer[128];

FILE
     *tdafvolt, *tiafvolt,
     *dsrdvolt, *tdafvec,
     *tiafvec, *numbers;

if ((tdafvolt = fopen("tdafvolt.dat", "w")) == NULL)
   printf (" *** Could not open tdafvolt.dat! *** \n");

if ((tiafvolt = fopen("tiafvolt.dat", "w")) == NULL)
   printf (" *** Could not open tiafvolt.dat! *** \n");

if ((dsrdvolt = fopen("dsrdvolt.dat", "w")) == NULL)
   printf (" *** Could not open dsrdvolt.dat! *** \n");

if ((tdafvec = fopen("tdafvec.dat", "w")) == NULL)
   printf (" *** Could not open tdafvec.dat! *** \n");
```

```
   if ((tiafvec = fopen("tiafvec.dat", "w")) == NULL)
      printf (" *** Could not open tiafvec.dat! *** \n");

   if ((numbers = fopen("numbers.tex", "w")) == NULL)
      printf (" *** Could not open tdafnum.dat! *** \n");


/*************************************************************************
                    User Input Section
*************************************************************************/

   printf("Seed the random bit generator:  ");
   gets(buffer);
   sscanf(buffer, "%U", &bseed);
   printf("%d\n", bseed);

   printf("Seed the AWGN Generator (integer < 0): ");
   gets(buffer);
   sscanf(buffer, "%d", &nseed);
   printf("%d\n", nseed);

   printf("Number of samples per symbol required: ");
   gets(buffer);
   sscanf(buffer, "%d", &sampersym);
   printf("%d\n", sampersym);

   samplefreq = sampersym*datafreq;
   samplerate = (double) 1.0/samplefreq;

   printf(...anchester ( , ). Bipolar (1) format: ");
   gets(buffer);
   sscanf(buffer, "%d", &format);
   printf("%d\n", format);

   printf("Pulse shaping? (1=y, 0=n): ");
   gets(buffer);
   sscanf(buffer, "%d", &shape);
   printf("%d\n", shape);

   printf("Number of taps in the FIR filters: ");
   gets(buffer);
   sscanf(buffer, "%d", &numfirtaps);
   printf("%d\n", numfirtaps);

   numfirdelays = numfirtaps - 1;
```

```
printf("Number of taps in the TIAF adaptive filter: ");
gets(buffer);
sscanf(buffer, "%d", &numtiaftaps);
printf("%d\n", numtiaftaps);

numtiafdelays = numtiaftaps-1;

printf("Number of taps in the TDAF adaptive filter: ");
gets(buffer);
sscanf(buffer, "%d", &numtdaftaps);
printf("%d\n", numtdaftaps);

printf("SOI pulse shaping LPF cutoff (Hz): ");
gets(buffer);
sscanf(buffer, "%lf", &lpfin_cutoff);
printf("%f\n", lpfin_cutoff);

ilpfin_cutoff = lpfin_cutoff;

printf("Gain of pulse shaping LPF: ");
gets(buffer);
sscanf(buffer, "%lf", &gain1);
printf("%f\n", gain1);

igain1 = gain1;

printf("SOI carrier amplitude: ");
gets(buffer);
sscanf(buffer, "%lf", &carrierampl);
printf("%f\n", carrierampl);

desiredpower = (sqr(carrierampl))/2;

printf("SOI carrier frequency: ");
gets(buffer);
sscanf(buffer, "%lf", &carrierfreq);
printf("%f\n", carrierfreq);

printf("Symbol frequency for the interferer:  ");
gets(buffer);
sscanf(buffer, "%lf", &idatafreq);
printf("%f\n",idatafreq);

idatarate = 1.0/idatafreq;
```

```
printf("SNOI carrier amplitude: ");
gets(buffer);
sscanf(buffer, "%lf", &icarrierampl);
printf("%f\n", icarrierampl);

interfpower = (sqr(icarrierampl))/2;

printf("SNOI carrier frequency: ");
gets(buffer);
sscanf(buffer, "%lf", &icarrierfreq);
printf("%f\n", icarrierfreq);

printf("Gain of output LPF: ");
gets(buffer);
sscanf(buffer, "%lf", &gain2);
printf("%f\n", gain2);

printf("Output LPF cutoff (Hz): ");
gets(buffer);
sscanf(buffer, "%lf", &lpfout_cutoff);
printf("%f\n", lpfout_cutoff);

printf("Phase shift for demodulator: ");
gets(buffer);
sscanf(buffer, "%lf", &deltaphase);
printf("%f\n", deltaphase);

printf("Noise factor: ");
gets(buffer);
sscanf(buffer, "%lf", &noisegain);
printf("%f\n", noisegain);

printf("Misadjustment factor: ");
gets(buffer);
sscanf(buffer, "%lf", &misadjust);
printf("%f\n", misadjust);

if (carrierfreq > 0.0)
   mu = misadjust/((desiredpower + interfpower +
        sqr(noisegain))*(numtdaftaps));
else
   mu = misadjust/((sqr(carrierampl) + sqr(icarrierampl) +
        sqr(noisegain))*(numtdaftaps));
```

```
   fprintf(numbers,"Mu = %f\n", mu);
   printf("Random data (1) or square wave (0): ");
   gets(buffer);
   sscanf(buffer, "%d", &outputflag);
   printf("%d\n", outputflag);

   printf("Number of symbols in one epoch: ");
   gets(buffer);
   sscanf(buffer, "%u", &numsymbols);
   printf("%d\n", numsymbols);

   numsamples = numsymbols*sampersym;

   adaploops = ((adapfactor/mu)/numsymbols);

   printf("Number of epochs to average: ");
   gets(buffer);
   sscanf(buffer, "%d", &avgloops);
   printf("%d\n", avgloops);

   printf("Number of adapation epochs: ");
   gets(buffer);
   sscanf(buffer, "%d", &adaploops);
   printf("%d\n", adaploops);

   numloops = adaploops + avgloops;
   fprintf(numbers,"Number of adaptation epochs: %d\n", adaploops);
   fprintf (numbers,"Total number of epochs: %d\n", numloops);
   printf("Total number of epochs: %d\n", numloops);
   fprintf (numbers, "Total number of post-adaptation symbols: %d\n",
                    numsymbols*avgloops);
   loopfactor = 1.0/(numloops);

/*************************************************************************
                    Initialization Section
*************************************************************************/

   if (shape == 1) {
      lpfconts       = dvector(0, numfirdelays);
      lpfwts       = dvector(0, numfirdelays);
      ilpfconts = dvector(1, numfirtaps);
      ilpfwts = dvector(1, numfirtaps);
      omega_c = setcutoff(samplerate, lpfin_cutoff);
      iomega_c = setcutoff(samplerate, ilpfin_cutoff);
      calcfilterweights (numfirdelays, omega_c, lpfconts, lpfwts);
```

```
    calcfilterweights (numfirdelays, iomega_c, ilpfconts, ilpfwts);
  }

  if (carrierfreq > 0.0) {
    outweights       = dvector(0, numfirdelays);
    outcontents      = dvector(0, numfirdelays);
    tiafoutweights     = dvector(0, numfirdelays);
    tiafoutcontents    = dvector(0, numfirdelays);
    nnwts = dvector(0, numfirdelays);
    nnconts = dvector(0, numfirdelays);
    omega_c1 = setcutoff(samplerate, lpfout_cutoff);
    calcfilterweights (numfirdelays, omega_c1, outcontents,
                       outweights);
    calcfilterweights (numfirdelays, omega_c1, tiafoutcontents,
                       tiafoutweights);
    calcfilterweights (numfirdelays, omega_c1, nnconts, nnwts);
  }

  tiafwts       = dvector(0, numtiafdelays);
  tiafconts       = dvector(0, numtiafdelays);

  mse = vector(0, numsamples-1);
  tempmse = vector(0, numsamples-1);
  tiafmse = vector(0, numsamples-1);
  tiaftempmse = vector(0, numsamples-1);

  for (sample = 0; sample < numsamples; ++sample) {
    mse[sample] = 0.0;
    tempmse[sample] = 0.0;
    tiafmse[sample] = 0.0;
    tiaftempmse[sample] = 0.0;
  }

  inittiaf(tiafwts, tiafconts, numtiaftaps, &tiaferror, &tiafbias);
  tdafwts = inittdafwts(sampersym, numtdaftaps);
  tdafconts = inittdafconts(numtdaftaps);
  tdaferror = initerror(sampersym);
  tdafbias = initbias(sampersym);
  time = ran1(&nseed);
  lasttime = time-datarate;
  itime = ran1(&nseed);
  ilasttime = itime-idatarate;

/*****************************************************************
                    Simulation Section
```

```
**********************************************************************/

   for (loopnum = 1; loopnum <= numloops; ++loopnum){
      for (sample = 0; sample < numsamples; ++sample) {

/**********************************************************************
              Interference (SNOI) Section
**********************************************************************/
         if (icarrierampl > 0.0) {
            d1t = datagen(itime, idatarate, d1t, &ilasttime,
                  &bseed, outputflag);

            if (format == 0)
               d2t = manchester(d1t, itime, idatarate, ilasttime);
            else
               d2t = bipolar(d1t);

            if (shape == 1)
               s1t = lpf(d2t, numfirdelays, ilpfwts, ilpfconts, igain1);
            else
               s1t = d2t;

            if (carrierfreq > 0.0)
               s2t = modulate(s1t, icarrierfreq, icarrierampl, itime);
            else
               s2t = s1t;
         }
         else
            s2t = 0.0;

/**********************************************************************
                     Signal (SOI) Section
**********************************************************************/

         d3t = datagen(time, datarate, d3t, &lasttime,
               &bseed, outputflag);

         if (format == 0)
            d4t = manchester(d3t, time, datarate, lasttime);
         else
            d4t = bipolar(d3t);
         s3tlast = s3t;
         if (shape == 1)
            s3t = lpf(d4t, numfirdelays, lpfwts, lpfconts, gain1);
         else
```

```
      s3t = d4t;

   if (carrierfreq > 0.0)
      s4t = modulate(s3t, carrierfreq, carrierampl, time);
   else
      s4t = s3t;

/*************************************************************************
                    Channel Section
*************************************************************************/
   s5t = s4t  + s2t + gasdev(&nseed)*noisegain;

/*************************************************************************
                TDAF Receiver Section
*************************************************************************/

    s6t = tdaf(s5t, s4t, mu, sampersym, numtdaftaps, tdafwts,
               tdafconts, tdaferror, tdafbias, frozen);

   if (carrierfreq > 0.0 )
      s7t = demodulate( s6t, carrierfreq, deltaphase, time);
      s8t = lpf(s7t, numfirdelays, outweights, outcontents,
               gain2);
   }
   else
      s8t = s6t;

/*************************************************************************
                TIAF Receiver Section
*************************************************************************/
   x6t = tiaf(s5t, s4t, mu, numtiaftaps, &tiaferror,
         tiafwts, tiafconts, &tiafbias, frozen);

   if (carrierfreq > 0.0) {
      x7t = demodulate(x6t, carrierfreq, deltaphase, time);
      x8t = lpf(x7t, numfirdelays, tiafoutweights,
            tiafoutcontents, gain2);
   }
   else
      x8t = x6t;

/*************************************************************************
                Noise Free Section
*************************************************************************/
```

```
        nn2tlast = nn2t;
        if (carrierfreq > 0.0) {
           nn1t = demodulate(s4t, carrierfreq, deltaphase, time);
           nn2t = lpf(nn1t, numfirdelays, nnwts, nnconts, gain2);
        }
        else
           nn2t = s4t;

        time += samplerate;
        itime += samplerate;

/*************************************************************************
                  End of Receiver section
*************************************************************************/
        if ((!offsetfound) && (loopnum == adaploops +1)){
           if (((nn2t > 0.0) && (nn2tlast < 0.0)) ||
               ((nn2t < 0.0) && (nn2tlast > 0.0)))  {
              bitvar = 0;
              tdafbitnum = 0;
              tiafbitnum = 0;
              dsrdbitnum = 0;
              offsetfound = 1;
           }
        }


        if (loopnum == numloops) {
           fprintf(dsrdvolt, "%d %f\n", sample, nn2t);
           fprintf(tdafvolt, "%i %f\n", sample, s8t);
           fprintf(tiafvolt, "%i %f\n", sample, x8t);
        }

        if (loopnum > adaploops) {
           frozen = 1;
           tempmse[sample]=sqr(s4t-s6t)/desiredpower;
           tiaftempmse[sample]=sqr(s4t-x6t)/desiredpower;
           if (format) {
              ++bitvar;
              tdafbitnum += s8t;
              tiafbitnum += x8t;
              dsrdbitnum += nn2t;
              if (!(bitvar % sampersym)) {
                 if (((tdafbitnum > 0.0) && (dsrdbitnum < 0.0)) ||
                     ((tdafbitnum < 0.0) && (dsrdbitnum > 0.0)))
                    ++tdafbiterror;
```

```
                if (((tiafbitnum > 0.0) && (dsrdbitnum < 0.0)) ||
                    ((tiafbitnum < 0.0) && (dsrdbitnum > 0.0)))
                   ++tiafbiterror;
                ++bitcount;
                tdafbitnum = 0.0;
                tiafbitnum = 0.0;
                dsrdbitnum = 0.0;
             }
          }   /* end  if(format) */
       }   /* end if(loopnum>adaploops) */
    }   /* Ends inner FOR loop */

    if (loopnum > adaploops) {
       ++msecount;
       for (sample1 = 0; sample1 < numsamples; ++sample1) {
          mse[sample1]+=tempmse[sample1];
          tiafmse[sample1]+=tiaftempmse[sample1];
       }
    }

} /* Ends outter FOR loop */

for (sample = 0; sample < numsamples; ++sample) {
   fprintf(tdafvec,"%i %f\n", sample, (float) mse[sample]/msecount);
   fprintf(tiafvec,"%i %f\n", sample,
                                  (float) tiafmse[sample]/msecount );
}
avg = 0.0;
tiafavg = 0.0;
count = 0;
for (sample = 0; sample < numsamples; ++sample) {
   avg += (double) mse[sample]/msecount;
   tiafavg += (double) tiafmse[sample]/msecount;
   ++count;
}
avg/=count;
tiafavg/=count;
fprintf(numbers,"average error for TDAF: %f\n", avg);
fprintf(numbers,"average error for TIAF: %f\n", tiafavg);
fprintf(numbers,"number of tdaf bit errors: %d ", tdafbiterror);
fprintf(numbers,"for a bit error rate of %f\n", (float)
                tdafbiterror/bitcount);
fprintf(numbers,"number of tiaf bit errors: %d ", tiafbiterror);
fprintf(numbers,"for a bit error rate of %f\n", (float)
                tiafbiterror/bitcount);
```

```
   fclose(tdafvolt);
   fclose(tiafvolt);
   fclose(dsrdvolt);
   fclose(tdafvec);
   fclose(tiafvec);
   fclose(numbers);

return 0;

} /* Ends main() */

/*************************************************************/

double setcutoff(double samplerate,  double cutoff)  {

   double temp = (2 * pi * cutoff * samplerate);
   return temp;
}

/*************************************************************/

void calcfilterweights(int numdelays, double omega_c,
         double *contents, double *weights) {

   int count = -1;
   double normfactor = 1.0;
   double temp, window;
   double *contents_ptr = contents;
   double *weight_ptr = weights;
   double *array_end = weights + numdelays + 1;
   double M = numdelays;
   for ( weight_ptr = weights; weight_ptr < array_end; ++weight_ptr ) {

      count += 1;
      *contents_ptr = 1.0;
      ++contents_ptr;
      temp = count - (M/2);
      window = 0.54 - 0.46*cos(2*pi*count/(M));

      if (temp == 0) {
         *weight_ptr = window*normfactor;
         }
      else
```

```
            *weight_ptr = window*(sin(omega_c*temp))/(pi*temp)*normfactor;
      }

}


/*************************************************************/


#define IB1 1
#define IB2 2
#define IB5 16
#define IB18 131072
#define MASK IB1+IB2+IB5

int irbit2(unsigned long *bseed) {

      if (*bseed & IB18) {

         *bseed = (( *bseed ^ MASK) << 1) | IB1;
         return 1;
      }
      else {

         *bseed <<= 1;
         return 0;
      }
}


/*************************************************************/

int datagen(double time, double datarate, int lastdata,  double
                *lasttime, unsigned long *bseed, int outputflag)  {

      int temp;
      if (time >= (*lasttime + datarate)) {
         *lasttime = time;
         if (outputflag == 1) {
            temp = irbit2(bseed);
          }
         else
            temp = -lastdata;
      }
      else
```

```
      temp = lastdata;
   return temp;

}

/*************************************************************/

double lpf(double input, int numdelays, double *weights,
          double *contents, double gain) {

   int count;
   double sum = 0;
   double *contents_end = contents + (numdelays);
   double *weights_end = weights + (numdelays);

   for (count = numdelays; count > 0; --count) {

      *(contents_end)  = *(contents_end - 1);
      sum += *weights_end * *contents_end;
      --contents_end;
      --weights_end;
      }

   *contents = input;
   sum += *weights * *contents;
   return sum * gain;

}


/*************************************************************/


double modulate(double input, double carrierfreq,
               double carrierampl, double time) {


   double temp = input*carrierampl*cos(2*pi*carrierfreq*time);
   return temp;

}

/*************************************************************/
```

```
double demodulate(double input, double carrierfreq,
              double phase, double time) {

   double temp = 2 * input * cos(2 * pi * carrierfreq *
                 time + phase);
   return temp;

}

/*************************************************************/
/* The following routine was taken from
"Numerical Recipes in C" by Press et al. */

#define M1 259200
#define IA1 7141
#define IC1 54773
#define RM1 (1.0/M1)
#define M2 134456
#define IA2 8121
#define IC2 28411
#define RM2 (1.0/M2)
#define M3 243000
#define IA3 4561
#define IC3 51349

float ran1(int *idum) {


   static long ix1,ix2,ix3;
   static float r[98];
   float temp;
   static int iff=0;
   int j;
   void nrerror();

   if (*idum < 0 || iff == 0) {
      iff=1;
      ix1=(IC1-(*idum)) % M1;
      ix1=(IA1*ix1+IC1) % M1;
      ix2=ix1 % M2;
      ix1=(IA1*ix1+IC1) % M1;
      ix3=ix1 % M3;
      for (j=1;j<=97;j++) {
         ix1=(IA1*ix1+IC1) % M1;
         ix2=(IA2*ix2+IC2) % M2;
```

```
            r[j]=(ix1+ix2*RM2)*RM1;
         }
         *idum=1;
      }
      ix1=(IA1*ix1+IC1) % M1;
      ix2=(IA2*ix2+IC2) % M2;
      ix3=(IA3*ix3+IC3) % M3;
      j=1 + ((97*ix3)/M3);
      if (j > 97 || j < 1) nrerror("RAN1: This cannot happen.");
      temp=r[j];
      r[j]=(ix1+ix2*RM2)*RM1;
      return temp;
   }

   #undef M1
   #undef IA1
   #undef IC1
   #undef RM1
   #undef M2
   #undef IA2
   #undef IC2
   #undef RM2
   #undef M3
   #undef IA3
   #undef IC3


   /*************************************************************/
   /* The following routine was taken from
   "Numerical Recipes in C" by Press et al. */

   float gasdev(int *idum) {

      static int iset=0;
      static float gset;
      float fac,r,v1,v2;
      float ran1();

      if  (iset == 0) {
         do {
            v1=2.0*ran1(idum)-1.0;
            v2=2.0*ran1(idum)-1.0;
            r=v1*v1+v2*v2;
         } while (r >= 1.0 || r == 0.0);
         fac=sqrt(-2.0*log(r)/r);
```

```
      gset=v1*fac;
      iset=1;
      return v2*fac;
   } else {
      iset=0;
      return gset;
   }
}

/*********************************************************/

double **inittdafwts(int sampersym, int numtaps) {

   int rownum, colnum;
   double **tdafwts;
   tdafwts = dmatrix(1, sampersym, 1, numtaps);

   for (rownum = 1; rownum <= sampersym; ++rownum)

         for (colnum = 1; colnum <= numtaps; ++colnum)

            tdafwts[rownum][colnum] = 0.0;


   return tdafwts;

}
/*********************************************************/

double *inittdafconts(int numtaps) {

   int tapnum;
   double *tdafconts;
   tdafconts = dvector(1, numtaps);
   for (tapnum = 1; tapnum <= numtaps; ++tapnum)
      tdafconts[tapnum] = 0.0;
   return tdafconts;
}
/* *********************************************************/

double tdaf(double input, double desired, double mu,
   int sampersym, int numtaps, double **tdafwts,
   double *tdafconts, double *error, double *bias, int frozen) {

   /*********************************************************
```

```
/ This routine is the Time Dependent Adaptive Filter.
/
/   Input Parameters:
/
/      input:      The value to be filtered.
/      desired:    The training signal.
/      mu:         Used to calculate the filter weights.
/      sampersym:  The number of samples per symbol of
/            the input baseband data sequence.
/            The TDAF is made up of sampersym
/            TIAFs.
/      numtaps:    The number of taps in each of the
/            member TIAFs making up the TDAF.
/      tdafwts:    An n by m array where n = numtaps and
/            m = sampersym containing the current
/            filter coefficients of the TDAF.
/      tdafconts:  A vector that holds the past values
/            of the input.
/      error:      A vector of length
/            sampersym that contains the quantity
/            "desired - sum" (see sum below) for each
/            TIAF in the TDAF.
/
/   Local Variables:
/
/      callnum:    Used to calculate which TIAF supplies
/            the output for each call to the function
/      tapnum:     Indexing variable for the taps of each
/            TIAF.
/      filternum: The index that points to the TIAF
/            that supplies the output of the function.
/      sum:        Holds the intermediate and final value
/            of the output of the function.
/
*************************************************************/

static int callnum = 0;
int tapnum, filternum;
double sum = 0.0;
filternum = (callnum % sampersym) + 1;
++callnum;
/* Shift data in filter to the right... */
for (tapnum = numtaps; tapnum > 1; --tapnum) {
   tdafconts[tapnum] = tdafconts[tapnum - 1];
   /* and calculate the output due to the past values */
```

```
      sum += tdafconts[tapnum] * tdafwts[filternum][tapnum];
   }
   /* Now shift the input into tap number 1 */
   tdafconts[1] = input;
   sum += tdafconts[1] * tdafwts[filternum][1];
   /* Add the contribution due to the bias weight */
   sum += bias[filternum];
   /* Calculate the error */
   error[filternum] = desired - sum;
   /* Update the filterweights for the next call */
   if (!frozen) {
      for (tapnum = 1; tapnum <= numtaps; ++tapnum) {
         tdafwts[filternum][tapnum] += 2 * mu * error[filternum]
                     * tdafconts[tapnum];
      }
      /* Update the bias weight */
      bias[filternum] += 2 * mu * error[filternum];
   }
   return sum;
}


/*************************************************************/

double *initerror(int sampersym) {

   int filternum;
   double *error;
   error = dvector(1, sampersym);
   for (filternum = 1; filternum <= sampersym; ++filternum)
      error[filternum] = 0.0;
   return error;
}


/*************************************************************/

double *initbias(int sampersym) {

   int filternum;
   double *bias;
   bias = dvector(1, sampersym);
   for (filternum=1;filternum<=sampersym;++filternum)
      bias[filternum] = 0.0;
   return bias;
```

```
}

/*********************************************************************/
/* The following routine was taken from
"Numerical Recipes in C" by Press et al. */

void nrerror(char error_text[]) {

   fprintf(stderr,"Numerical Recipes run-time error...\n");
   fprintf(stderr,"%s\n",error_text);
   fprintf(stderr,"...now exiting to system...\n");
   exit(1);
}


/*********************************************************************/
/* The following routine was taken from
"Numerical Recipes in C" by Press et al. */

double *dvector(int nl, int nh) {

   double *v;

   v=(double *)malloc((unsigned) (nh-nl+1)*sizeof(double));
   if (!v) nrerror("allocation failure in dvector()");
   return v-nl;
}

/*********************************************************************/
/* The following routine was taken from
"Numerical Recipes in C" by Press et al. */

double **dmatrix(int nrl, int nrh, int ncl, int nch) {

   int i;
   double **m;

   m=(double **) malloc((unsigned) (nrh-nrl+1)*sizeof(double*));
   if (!m) nrerror("allocation failure 1 in dmatrix()");
   m -= nrl;

   for(i=nrl;i<=nrh;i++) {
      m[i]=(double *) malloc((unsigned) (nch-ncl+1)*sizeof(double));
      if (!m[i]) nrerror("allocation failure 2 in dmatrix()");
      m[i] -= ncl;
```

```
    }
    return m;
}

/*************************************************************/

int manchester(int input, double time, double datarate, double lasttime) {
   int temp1, temp2;
   if ( time > (lasttime + datarate/2.0))
      temp1 = abs(input-1);
   else
      temp1 = input;
   if (temp1 == 1)
      temp2 = 1;
   else
      temp2 = -1;
   return temp2;
}

/*************************************************************/
/* The following routine was taken from
"Numerical Recipes in C" by Press et al. */

void free_dmatrix(double **m, int nrl, int nrh, int ncl, int nch)

{
   int i;

   for(i=nrh;i>=nrl;i--) free((char*) (m[i]+ncl));
   free((char*) (m+nrl));
}

/*************************************************************/
/* The following routine was taken from
"Numerical Recipes in C" by Press et al. */

void free_dvector(double *v, int nl, int nh)

{
   free((char*) (v+nl));
}

/*************************************************************/
/* The following routine was taken from
"Numerical Recipes in C" by Press et al. */
```

```
float *vector(int nl,int nh) {

   float *v;

   v=(float *)malloc((unsigned) (nh-nl+1)*sizeof(float));
   if (!v) nrerror("allocation failure in vector()");
   return v-nl;
}

/*************************************************************/

void inittiaf (double *weights, double *contents, int numtaps,
               double *error, double *bias) {

   int count;
   *error = 0.0;
   *bias = 0.0;
   for (count = 0; count < numtaps; ++count) {

      *(weights + count) = 0;
      *(contents + count) = 0;
      }
}

/*************************************************************/

double tiaf      (float input, float desired, float mu, int numtaps,
                        double *error, double *weights, double
                        *contents, double *bias, int frozen) {

   int numdelays = numtaps - 1;
   int count;
   double sum = 0.0;

   for (count = numdelays; count > 0; --count)  {

      contents[count] = contents[count - 1];
      sum += contents[count] * weights[count];

      }

   contents[0] = input;
   sum += contents[0] * weights[0];
   sum += *bias;
```

```
   *error = desired - sum;

   if (!frozen) {
      for (count = 0; count < numtaps; ++count)
         weights[count] += 2 * mu * *error * contents[count];
      *bias += 2 * mu * *error;
   }
   return sum;

}

/*************************************************************/

int bipolar(int input) {

   if ( input == 1 )
      return 1;
   else
      return -1;
}
/*************************************************************/
```

Appendix D. *Source Code for the LC Version*

```
/*****************************************************************/
/**** This program is a simulated digital communication system *******/
/*****************************************************************/
/*  30 Sep 91:  Made number of symbols for simulation user selectable */
/   Executable file:  lrncrv                                        /
/*****************************************************************/
#define sparc 1

#include <stdio.h>
#include <math.h>
#include <stdlib.h>

/*********************************************************/

#define pi                (4*atan(1.0))
#define data_freq         1.0
#define data_rate         1.0/datafreq
#define sqr(x)            (x)*(x)

/*********************************************************/

/* Function Prototypes */

/* Refer to Appendix A - All Functions are the same  */

/*********************************************************/

int main() {


   unsigned long numsamples;
   int numfirtaps, numtiaftaps, numtdaftaps;
   int    numfirdelays, numtiafdelays;
   int    numloops;
   int    samplefreq;
   int    count;
   int   outputflag;
   int   numsym;
   float  loopfactor;
   double samplerate;
```

```
double datafreq    = data_freq;
double datarate    = data_rate;
double carrierfreq;
double carrierampl;
double noisegain;
double misadjust;
double mu;
double deltaphase;
double gain1;
double gain2;
double tiafgain2;
double avg = 0.0;
double tiafavg = 0.0;
double lpfin_cutoff, lpfout_cutoff, ilpfin_cutoff;
char   buffer[128];

double *lpfconts,  *lpfwts;
double *nnwts, *nnconts;
double *tiafwts, *tiafconts;
double *outweights, *outcontents;
double *tiafoutweights, *tiafoutcontents;
double *mse, *tempmse;
double *tiafmse, *tiaftempmse;
double tiaferror = 0.0;
double tiafbias = 0.0;
int   sample, sample1, loopnum;
int d1t=1, d2t, d3t=1, d4t, d5t, d6t;
int   sampersym;
int format;
int shape;
double omega_c;
double omega_c1;
double time = 0.0;
double s1t, s2t, s3t, s4t, s5t, s6t, s7t, s8t;
double x6t, x7t, x8t;
double nn1t, nn2t;
unsigned long int bseed;

int nseed;
double lasttime = -datarate;

double **tdafwts;
double *tdafconts;
double *tdaferror;
double *tdafbias;
```

```
double *tdafsymbolmse;
double *tiafsymbolmse;
int msecount;
int msesize;
int mseindex = 0;
int numsymbols;
double idatarate;
double idatafreq;
double ilasttime;
unsigned long int ibseed;
double *ilpfconts, *ilpfwts;
double igain1;
double icarrierfreq;
double icarrierampl;
double itime = 0.25;
double iomega_c;
float adapfactor = 0.1178;
double desiredpower;
double interfpower;
int loopcount = 0;

FILE *tdaflrn, *tiaflrn, *lnumbers;

if ((tdaflrn = fopen("tdaflrn.dat", "w")) == NULL)
   printf (" *** Could not open tdaflrn.dat! *** \n");

if ((tiaflrn = fopen("tiaflrn.dat", "w")) == NULL)
   printf (" *** Could not open tiaflrn.dat! *** \n");

if ((lnumbers = fopen("lnumbers.tex", "w")) == NULL)
   printf (" *** Could not open lnumbers.tex! *** \n");


printf("Seed the random bit generator:  ");
gets(buffer);
sscanf(buffer, "%U", &bseed);
printf("%d\n", bseed);

printf("Seed the AWGN Generator (integer < 0): ");
gets(buffer);
sscanf(buffer, "%d", &nseed);
printf("%d\n", nseed);

printf("Number of samples per symbol required: ");
gets(buffer);
```

```
sscanf(buffer, "%d", &sampersym);
printf("%d\n", sampersym);

samplefreq = sampersym*datafreq;
samplerate = (double) 1.0/samplefreq;

printf("Manchester (0) or Bipolar (1) format: ");
gets(buffer);
sscanf(buffer, "%d", &format);
printf("%d\n", format);

printf("Pulse shaping? (1=y, 0=n): ");
gets(buffer);
sscanf(buffer, "%d", &shape);
printf("%d\n", shape);

printf("Number of taps in the FIR filters: ");
gets(buffer);
sscanf(buffer, "%d", &numfirtaps);
printf("%d\n", numfirtaps);

numfirdelays = numfirtaps - 1;

printf("Number of taps in the TIAF adaptive filter: ");
gets(buffer);
sscanf(buffer, "%d", &numtiaftaps);
printf("%d\n", numtiaftaps);

numtiafdelays = numtiaftaps-1;

printf("Number of taps in the TDAF adaptive filter: ");
gets(buffer);
sscanf(buffer, "%d", &numtdaftaps);
printf("%d\n", numtdaftaps);

printf("SOI pulse shaping LPF cutoff (Hz): ");
gets(buffer);
sscanf(buffer, "%lf", &lpfin_cutoff);
printf("%f\n", lpfin_cutoff);

ilpfin_cutoff = lpfin_cutoff;

printf("Gain of pulse shaping LPF: ");
gets(buffer);
sscanf(buffer, "%lf", &gain1);
```

```
printf("%f\n", gain1);

igain1 = gain1;

printf("SOI carrier amplitude: ");
gets(buffer);
sscanf(buffer, "%lf", &carrierampl);
printf("%f\n", carrierampl);

desiredpower = sqr(carrierampl)/2;

printf("SOI carrier frequency: ");
gets(buffer);
sscanf(buffer, "%lf", &carrierfreq);
printf("%f\n", carrierfreq);

printf("Symbol frequency for the interferer:  ");
gets(buffer);
sscanf(buffer, "%lf", &idatafreq);
printf("%f\n",idatafreq);

idatarate = 1.0/idatafreq;

printf("SNOI carrier amplitude: ");
gets(buffer);
sscanf(buffer, "%lf", &icarrierampl);
printf("%f\n", icarrierampl);

interfpower = sqr(icarrierampl)/2;

printf("SNOI carrier frequency: ");
gets(buffer);
sscanf(buffer, "%lf", &icarrierfreq);
printf("%f\n", icarrierfreq);

printf("Gain of output LPF: ");
gets(buffer);
sscanf(buffer, "%lf", &gain2);
printf("%f\n", gain2);

printf("Output LPF cutoff (Hz): ");
gets(buffer);
sscanf(buffer, "%lf", &lpfout_cutoff);
printf("%f\n", lpfout_cutoff);
```

```
printf("Phase shift for demodulator: ");
gets(buffer);
sscanf(buffer, "%lf", &deltaphase);
printf("%f\n", deltaphase);

printf("Noise factor: ");
gets(buffer);
sscanf(buffer, "%lf", &noisegain);
printf("%f\n", noisegain);

printf("Misadjustment factor: ");
gets(buffer);
sscanf(buffer, "%lf", &misadjust);
printf("%f\n", misadjust);

if (carrierfreq > 0.0)
   mu = misadjust/((desiredpower + interfpower +
        sqr(noisegain))*(numtdaftaps));
else
   mu = misadjust/((sqr(carrierampl) + sqr(icarrierampl) +
        sqr(noisegain))*(numtdaftaps));

printf("Mu = %.10g\n", mu);
fprintf(lnumbers,"Mu = %.10g\n", mu);

printf("Random data (1) or square wave (0): ");
gets(buffer);
sscanf(buffer, "%d", &outputflag);
printf("%d\n", outputflag);

printf("Number of epochs: ");
gets(buffer);
sscanf(buffer, "%d", &numloops);
printf("%d\n", numloops);

printf("Number of symbols to average: ");
gets(buffer);
sscanf(buffer, "%d", &numsym);
printf("%d\n", numsym);

printf("Number of symbols in simulation: ");
gets(buffer);
sscanf(buffer, "%d", &numsymbols);
printf("%d\n", numsymbols);
```

```
numsamples = numsymbols * sampersym;
fprintf(lnumbers,"Number of symbols in learing curve: %d\n",
                                    numsymbols);

loopfactor = 1.0/(numloops);

if (shape == 1) {
   lpfconts       = dvector(0, numfirdelays);
   lpfwts       = dvector(0, numfirdelays);
   ilpfconts = dvector(1, numfirtaps);
   ilpfwts = dvector(1, numfirtaps);
   omega_c = setcutoff(samplerate, lpfin_cutoff);
   iomega_c = setcutoff(samplerate, ilpfin_cutoff);
   calcfilterweights (numfirdelays, omega_c, lpfconts, lpfwts);
   calcfilterweights (numfirdelays, iomega_c, ilpfconts, ilpfwts);
}

if (carrierfreq > 0.0) {
   outweights       = dvector(0, numfirdelays);
   outcontents       = dvector(0, numfirdelays);
   tiafoutweights       = dvector(0, numfirdelays);
   tiafoutcontents    = dvector(0, numfirdelays);
   nnwts  = dvector(0, numfirdelays);
   nnconts = dvector(0, numfirdelays);
   omega_c1 = setcutoff(samplerate, lpfout_cutoff);
   calcfilterweights (numfirdelays, omega_c1, outcontents,
                      outweights);
   calcfilterweights (numfirdelays, omega_c1, tiafoutcontents,
                      tiafoutweights);
   calcfilterweights (numfirdelays, omega_c1, nnconts, nnwts);
}

tiafwts       = dvector(0, numtiafdelays);
tiafconts       = dvector(0, numtiafdelays);
msesize = numsamples/sampersym;
mse = dvector(0, msesize - 1);
tempmse = dvector(0, msesize - 1);
tiafmse = dvector(0, msesize - 1);
tiaftempmse = dvector(0, msesize - 1);
tdafsymbolmse = dvector(0, sampersym-1);
tiafsymbolmse = dvector(0, sampersym-1);

for (sample = 0; sample < msesize; ++sample) {
   mse[sample] = 0.0;
   tempmse[sample] = 0.0;
```

```
      tiafmse[sample] = 0.0;
      tiaftempmse[sample] = 0.0;
   }

   for (loopnum = 1; loopnum <= numloops; ++loopnum){
      inittiaf(tiafwts, tiafconts, numtiaftaps, &tiaferror, &tiafbias);
      tdafwts = inittdafwts(sampersym, numtdaftaps);
      tdafconts = inittdafconts(numtdaftaps);
      tdaferror = initerror(sampersym);
      tdafbias = initbias(sampersym);
      msecount = 0;
      time = ran1(&nseed);
      lasttime = time-datarate;
      itime = ran1(&nseed);
      ilasttime = itime-idatarate;

      for (sample = 0; sample < numsamples; ++sample) {

/*************************************************************
              Interference (SNOI) Section
*************************************************************/
         if (icarrierampl > 0.0) {
            d1t = datagen(itime, idatarate, d1t, &ilasttime,
                  &ibseed, outputflag);

            if (format == 0)
               d2t = manchester(d1t, itime, idatarate, ilasttime);
            else
               d2t = bipolar(d1t);

            if (shape == 1)
               s1t = lpf(d2t, numfirdelays, ilpfwts, ilpfconts, igain1);
            else
               s1t = d2t;

            if (carrierfreq > 0.0)
               s2t = modulate(s1t, icarrierfreq, icarrierampl, itime);
            else
               s2t = s1t;
         }
         else
            s2t = 0.0;

/*************************************************************
                    Signal (SOI) Section
```

```
**********************************************************/

        d3t = datagen(time, datarate, d3t, &lasttime,
              &bseed, outputflag);

        if (format == 0)
           d4t = manchester(d3t, time, datarate, lasttime);
        else
           d4t = bipolar(d3t);

        if (shape == 1)
           s3t = lpf(d4t, numfirdelays, lpfwts, lpfconts, gain1);
        else
           s3t = d4t;

        if (carrierfreq > 0.0)
           s4t = modulate(s3t, carrierfreq, carrierampl, time);
        else
           s4t = s3t;

/************************************************************
                      Channel Section
************************************************************/

        s5t = s4t  + s2t + gasdev(&nseed)*noisegain;

/************************************************************
                    TDAF Receiver Section
************************************************************/

         s6t = tdaf(s5t, s4t, mu, sampersym, numtdaftaps, tdafwts,
                    tdafconts, tdaferror, tdafbias);

        if (carrierfreq > 0) {
           s7t = demodulate(s6t, carrierfreq, deltaphase, time);
           s8t = lpf(s7t, numfirdelays, outweights, outcontents,
                     gain2);
        }
        else
           s8t = s6t;

/************************************************************
                    TIAF Receiver Section
************************************************************/
```

```
         x6t = tiaf(s5t, s4t, mu, numtiaftaps, &tiaferror,
               tiafwts, tiafconts, &tiafbias);

         if (carrierfreq > 0.0) {
            x7t = demodulate(x6t, carrierfreq, deltaphase, time);
            x8t = lpf(x7t, numfirdelays, tiafoutweights,
                  tiafoutcontents, gain2);
         }
         else
            x8t = x6t;

/*************************************************************
                  Noise Free Section
*************************************************************/
         if (carrierfreq > 0.0) {
            nn1t = demodulate(s4t, carrierfreq, deltaphase, time);
            nn2t = lpf(nn1t, numfirdelays, nnwts, nnconts, gain2);
         }
         else
            nn2t = s4t;
         time += samplerate;
         itime += samplerate;

         mseindex = sample % sampersym;
         tdafsymbolmse[mseindex] = (sqr(s4t-s6t))/(desiredpower);
         if (mseindex == (sampersym - 1)) {
            avg = 0.0;
            for (sample1 = 0; sample1 < sampersym; ++sample1) {
               avg += tdafsymbolmse[sample1];
            }
            tempmse[msecount] = avg/sampersym;
            ++msecount;
         }
         if (sample < msesize)
            tiaftempmse[sample] = (sqr(s4t - x6t))/(desiredpower);

      } /* Ends inner FOR loop */
      ++loopcount;
      for (sample1 = 0; sample1 < msesize; ++sample1) {
         mse[sample1]+=tempmse[sample1];
         tiafmse[sample1]+=tiaftempmse[sample1];
      }

      free_dmatrix(tdafwts, 1, sampersym, 1, numtdaftaps);
      free_dvector(tdafconts, 1, numtdaftaps);
```

```
      free_dvector(tdaferror, 1, sampersym);
      free_dvector(tdafbias, 1, sampersym);

   } /* Ends outter FOR loop */

   for (sample = 0; sample < msesize; ++sample) {
      fprintf(tdaflrn,"%i %f\n", sample, (float) mse[sample]/loopcount);
      fprintf(tiaflrn,"%i %f\n", sample,
                        (float) tiafmse[sample]*loopfactor);
   }
   avg = 0.0;
   tiafavg = 0.0;
   count = 0;
   for
      (sample = 0; sample < msesize-numsym; ++sample) {
         avg += mse[sample]/loopcount;
         tiafavg += tiafmse[sample]/loopcount;
         ++count;
   }
   avg/=count;
   tiafavg/=count;
   fprintf(lnumbers,"average error for TDAF over %d samples: %f\n",
                    numsym, avg);
   fprintf(lnumbers,"average error for TIAF over %d samples: %f\n",
                    numsym, tiafavg);

   fclose(tdaflrn);
   fclose(tiaflrn);
   fclose(lnumbers);

return 0;

} /* Ends main() */
```

Bibliography

1. Earl R. Ferrara, Jr. and Bernard Widrow. "The Time Sequenced Adaptive Filter," *IEEE Transactions on Acoustics, Speech and Signal Processing, ASSP - 29*(3) (June 1981).

2. Gardner, William A. "Exploitation of Spectral Redundancy in Cyclostationary Signals," *IEEE Signal Processing Magazine, 8*:14–36 (apr 1991).

3. Kennedy, K. J. and E. K. Koh. "Frequency-Reuse Interferency in TDMA/QPSK Satellite Systems." *Fifth International Conference on Digital Satellite Communications*. 99–107. March 1981.

4. Nicholson, David L., "Introduction to Cyclostationary Signal Processing." A white paper provided by the author, Mar 1991.

5. Oppenheim, Allan V. and Ronald W. Schafer. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1989.

6. Press, William H. and others. *Numerical Recipes in C: the Art of Scientific Computing*. New York: Cambridge University Press, 1989.

7. Reed, Jeffrey H. *Time-Dependent Adaptive Filters for Interference Rejection*. Ph.D. dissertation, Department of Electrical Engineering and Computer Science, University of California, Davis, Dec 1987.

8. Shanmugan, K. Sam and Arthur M. Breipohl. *Random Signals: Detection, Estimation, and Data Analysis*. New York: John Wiley and Sons, 1988.

9. Sklar, Bernard. *Digital Communications, Fundamentals and Applications*. Englewood Cliffs, NJ: Prentice Hall, 1988.

10. Widrow, Bernard and Samuel D. Stearns. *Adaptive Signal Processing*. Englewood Cliffs NJ: Prentice Hall, 1985.

11. Williams, Robert, "Course Notes - EENG 791, Advanced Digital Signal Processing." School of Engineering, Air Force Institute of Technology, (AU), Wright-Patterson AFB OH, March 1991.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1991	Master's Thesis

4. TITLE AND SUBTITLE

A Time-Dependent Adaptive Filter for Cochannel Interference Reduction

5. FUNDING NUMBERS

6. AUTHOR(S)

Matthew H. Foster, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GE/ENG/91D-19

9. SPONSORING, MONITORING AGENCY NAME(S) AND ADDRESS(ES)

FTC/DXSI, WPAFB OH 45433-6508

10. SPONSORING, MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This thesis presents a Time Dependent Adaptive Filter (TDAF) which exploits the cyclostationarity of digitally modulated communications signals and seeks to improve the Signal to Interference Ratio (SIR) and Signal to Noise Ratio (SNR) of such signals. The TDAF is imbedded in a computer simulation of a simple communication system consisting of a data source, data formatter, pulse shaping filter, BPSK modulator, and demodulator. In the simulation the TDAF and a Time Independent Adaptive Filter (TIAF) attempt to extract the Signal of Interest (SOI) from noise or interference. The criteria of Mean Squared Error (MSE) is used as the primary means to compare the performance of the two adaptive filters. Plots of MSE improvement in interference and MSE improvement in noise are presented. For the case of interference, the improvement is measured as a function of the baud rate of the intereference signal, and carrier frequency of the interference signal. It is shown that with respect to the TIAF, the TDAF provides up to 3 dB of improvement in a noisy evironment, and up to 12 dB of improvement in an environment characterized by strong interference. Bit Error Rates (BER) for several simulations are presented. The data indicate that signficant improvements in BER might also be expected when a TDAF is used in lieu of a TIAF.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Adaptive Filters, Cyclosationarity, Signal Processing, Simulation	119
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL